University of Piraeus
Department of Digital Systems
Postgraduate Programme in Big Data & Analytics

Leveraging Information-driven Storage in an NFV Marketplace in the 5G Ecosystem

MSc Thesis

Stavrianos Panagiotis
Registration Number: ME1732

Supervisor
Dimosthenis Kyriazis
Assistant Professor

University of Piraeus
Department of Digital Systems
Postgraduate Programme in Big Data & Analytics

Leveraging Information-driven Storage in an NFV Marketplace in the 5G Ecosystem

MSc Thesis

Supervisor
Dimosthenis Kyriazis
Assistant Professor

Consistency is the hallmark of the unimaginative

-Oscar Wilde-

Abstract

In the previous years, preliminary interest and discussions about a possible 5G standard have evolved into a full-fledged conversation that has captured the attention and imagination of researchers and engineers around the world. 5G networks expect to provide significant advances in network management compared to traditional mobile infrastructures by leveraging intelligence capabilities such as data analysis, prediction, pattern recognition and artificial intelligence. The telecommunications industry was one of the first to adopt data mining technology on the grounds that companies routinely generate and store enormous amounts of high-quality data, have a very large customer base, and operate in a rapidly changing and highly competitive environment. The ultimate goal of 5G era aims in automation and autonomy, where it's an incomplete strategy to continue to focus only on identifying patterns.

The digital transformation of network infrastructure through Network Function Virtualization (NFV) and Software Defined Networking (SDN) is anticipated to play a pivotal role with the respect to the commercialization of 5G. It is indubitable that the NFV comprises a key enabler in the composition of the 5G infrastructure. NFV provides the grounds for the efficient reformation of the existing mechanisms on the conformation of the networks and services, with Telecommunication Service Providers (TSPs) rapidly introducing new revenue-generating services with a wider range of service requirements than ever before. Promptly, the exposure and the advertising of the available services across heterogeneous networks necessitates the establishment of a marketplace for their promotion and management of the relationships in different domains. In the network-centric business, the identification and discovery on the notion of decisions in modifying the the network are the real heart of the operation. Additionally, data-mining algorithms can offer substantial advantages based on virtualized components from a NFV marketplace, targeting at TSPs who request the commercialization of new virtualized products rapidly. The incorporation of an NFV Marketplace paves the way for expansion of the relationships between the TSPs and their own enterprise customers beyond just connectivity services. New business-to-business products can be offered to the enterprise customers, who can purchase them on demand.

An NFV Marketplace bridges a plethora of gaps in the relationship of virtualized network infrastructure, cloud applications, orchestration, and commercialization tools through an ecosystem of partners as it enables the provisioning of new services, including third-party offerings and the personalization of bundles of services for different markets. On this concept, the increasing importance of the NFV Marketplace as a medium for electronic and business transactions has served as a driving force for the development of recommender systems, aiding the personalized service transactions. An important catalyst in this regard is the ease with which the NFV Marketplace enables users to provide feedback about preferences. In such cases, users are able to easily provide feedback and enhance the recommendation procedure.

The subject of this MSc thesis is the introduction, the development and the integration of a novel NFV Marketplace, serving the entire storage, utilization and recommendation lifecycle of NFV artifacts in the 5G ecosystem. Going beyond a conventional data store, the NFV Marketplace delivers dynamic added-value mechanisms from the support of the developers in the provision of innovative storage and mining capabilities from the demand and supply sides respectively, addressing the needs of the diverse stakeholders. Additionally, the integration of innovative recommendation algorithms are incorporated in the NFV Marketplace with the aim of engaging personalized artifact recommendations to the customers, as to allow them to delve more deeply into the available business services without having to perform search after search.

Keywords

5G Networks, Network Function Virtualization, Service Marketplace Platform, NoSQL Databases, Data Mining, Service Recommendation Engine, Item-Item Collaborative Filtering, Graph-based Recommender Systems

Acknowledgements

At the outset, I would like to express my sincere gratitude to my MSc thesis advisory committee, namely Assistant Professor Dr Dimosthenis Kyriazis, Associate Professor Dr Ilias Maglogiannis and Associate Professor Dr Andriana Prentza. More specially, I would like to thank and express my great appreciation to my advisor, Assistant Professor Dr Dimosthenis Kyriazis, for his support, advices and his scientific guidance throughout my work and my studies. Our academic interactions have always been beneficial and thought provoking. I am substantially thankful for all the professional experiences and opportunities that I acquired because of you, as well as the excellent collaboration. Your support, ideas, and feedback have been absolutely invaluable for my personal and academic progress. In the same manner, I want to express my gratitude to the Alexander S. Onassis Public Benefit Foundation as I am honoured to be one of the recipients of a Onassis Foundation-Scholarship, which supported this MSc thesis. Your contribution and support have been vital to develop my scientific ideas and feel part of the family of the Onassis Foundation. The provided assistance and the generous support is deemed priceless. Also, I would like to sincerely thank the Assistant Professor Dr Christos Doulkeridis, for his invaluable help and contribution in many parts of this MSc thesis. Your expertise, deep knowledge and perspective helped me in several parts of this work.

Contents

Page

List of Figures **x**

List of Tables **xi**

List of abbreviations **xiv**

1 Introduction **1**

1.0.1 Motivation and Research Challenges . . . 3

1.0.2 Thesis Contribution . . . 4

2 Fundamentals of 5G Networks **5**

2.1 Cloud Computing . . . 6

2.1.1 Service and Deployment Models . . . 7

2.1.2 Resources and Scalability . . . 8

2.2 Foundations of the SDN Approach . . . 10

2.2.1 Key Principles and Infrastructure Requirements of SDN . . . 10

2.2.2 The SDN Architecture . . . 12

2.2.3 Infrastructure Layer . . . 14

2.2.4 Control Layer . . . 15

2.2.5 Application Layer . . . 17

2.2.5.1 Programming Languages . . . 17

2.2.5.2 Network Applications . . . 20

2.3 Network Function Virtualization . . . 24

2.3.1 Hardware and Container Virtualization . . . 25

2.3.2 Network Function Virtualization Concepts . . . 27

2.3.3 NFV Management and Orchestration . . . 30

3 Review of NoSQL Databases **33**

3.1 Concepts of the Relational Model . . . 35

3.2 Transaction Processing and ACID Theorem . . . 36

3.3 Theorem of CAP Trade-off 39
3.3.1 The consensus problem 39
3.3.2 CAP Theorem 42
3.4 NoSQL Ecosystem 44
3.4.1 BASE Principles 47
3.4.2 Categorization of NoSQL Databases 49
3.4.2.1 Key-value Databases 49
3.4.2.2 Document-oriented Model Databases 51
3.4.2.3 Column Family Model Databases 52
3.4.2.4 Graph Model Databases 54
3.4.3 Concepts and characteristics of NoSQL Databases 56
3.4.3.1 Consistency 57
3.4.3.2 Replication 57
3.4.3.3 Concurrency 58
3.4.3.4 Partitioning 60
3.4.3.5 Query Capabilities 62

4 The NFV Marketplace 69
4.1 Related Work 70
4.2 Architecture Overview 71
4.2.1 Positioning of the NFV Marketplace in the 5G Infrastructure 72
4.2.2 Consistent Management of Container Artifacts 73
4.2.3 Internal Modules and Architecture of the NFV Marketplace 75
4.3 NFV Marketplace External Driver 76
4.3.1 User Interfaces for NFV Descriptor and Package Management 76
4.3.2 Attachment of Enriched Metadata 77
4.3.3 Full–text Search in Structure–Agnostic NFV Descriptors and Packages 79
4.3.4 Management of data redundancy 80
4.3.5 Decision Support Mechanism 81
4.3.6 Continuous Optimization 83
4.4 Numerical Results 84

5 Conclusions and Future Work 89
5.1 General Overview and Conclusions 89
5.2 Issues for future research 89

Bibliography 105

List of Figures

2.1 Vertical and horizontal scaling . . . 9
2.2 The SDN enablement of software-oriented and abstracted interaction in the network . . . 11
2.3 Control and data planes for (a) the traditional network architecture and (b) the SDN approach . 12
2.4 Software Defined Network Architecture . . . 13
2.5 Elements and interfaces of an SDN Controller . . . 15
2.6 Functionalities and components of the SDN application layer . . . 21
2.7 Types of Virtual Machine Monitors . . . 25
2.8 Architecture of container virtualization . . . 26
2.9 Notion of NFV from the perspective of NFV network appliance deployment (left) and high-level NFV architecture (right) . . . 27
2.10 NFV reference architectural framework . . . 28
2.11 NFV configuration of a network service. Top, graph representation of an end-to-end network service. Bottom, example of an end-to-end network service with VNFs and nested forwarding graph . . . 32

3.1 General structure of database management system . . . 34
3.2 Three schema approach . . . 35
3.3 Concurrent transaction processing in multiprogramming operating systems. Left, interleaved processing and right, parallel processing . . . 39
3.4 Visualization of CAP Theorem . . . 40
3.5 Visualization of CAP theorem on distributed database management systems from the perspective of (a) partition tolerance, (b) consistency and (c) availability . . . 43
3.6 Characteristics of NoSQL Databases . . . 47
3.7 Data structure in key-value databases . . . 50
3.8 Documents in a document-oriented database considering (a) an individual JSON sample with the key and its value and (b) the organization of the documents in a collection of the store . . . 52
3.9 Internal data structure in the columnar databases. On the left, the keyspace contains column families and on the right, a column family containing a variable number of rows, each one containing its own variable set of columns. . . . 53
3.10 Internal data storage of columns in columnar databases . . . 54

3.11 Building blocks of the property graph model 55
3.12 Example of a graph representation through (a) RDF and (b) property graph 55
3.13 Replication models of the (a) Master-slave and (b) Multi-master approach 59
3.14 Partitioning strategies of i. Range-based and ii. Consistent hash partitioning 61

4.1 Positioning of the NFV Marketplace inside the 5G environment 71
4.2 Generalized model of the NFV package format 74
4.3 Architecture and Internal Modules of the NFV Marketplace 75
4.4 Document management flow diagram in the NFV Marketplace 77
4.5 Schema of the initial attached metadata at the root level of each document 79
4.6 Composition of the inverted file 81
4.7 Management of the IF and full-text search mechanism in the NFV Marketplace 82
4.8 Incremental Item-to-Item Collaborative Filtering in the NFV Marketplace 82
4.9 Mean time performance of the POST operation of a single document as a function of the hosted number of documents for several depths of the JSON documents in the NFV Marketplace . . . 86
4.10 Minimum and maximum observations of the mean time performance of the POST operation as a function of the JSON Depth in the NFV Marketplace and MongoDB 87
4.11 Mean time performance of the PUT operation as a function of the JSON Depth in the NFV Marketplace and MongoDB 88

List of Tables

2.1 Comparison of horizontal and vertical scaling schemes 9
2.2 Programming languages in the SDN application layer 19

3.1 Set of business and technical requirements addressed from the NoSQL ecosystem 45
3.2 Terminology mapping of relational databases with the document-oriented databases 51
3.3 Comparison of property and RDF graphs 56
3.4 Partitioning, replication, consistency and concurrency mechanisms 63
3.5 Query features of NoSQL databases 66

4.1 Exposed endpoints for the Catalogues 78
4.2 Syntax of the URI string for querying the NFV Marketplace 80
4.3 Available operators implemented for querying the NFV Marketplace 83
4.4 Mean time performance of CRUD operations of NFV Marketplace and MongoDB loaded with 10^3 documents as a function of the JSON Depth 85
4.5 Mean time performance of full-text search of NFV Marketplace and MongoDB through indexed and unindexed fields as a function of the JSON Depth 87

List of abbreviations

NFV	Network Function Virtualization
SDN	Software-defined Networking
TSP	Telecommunication Service Provider
UHD	Ultra-high Definition
AR	Augmented Reality
IoE	Internet of Everything
ITU	International Telecommunication Union
IoT	Internet of Things
M2M	Machine to machine
NS	Network Service
CAPEX	Capital Expenditure
OPEX	Operation Expenditure
QoS	Quality of Service
QoE	Quality of Experience
IDS/IPS	Intrusion Detection/Prevention Systems
RAN	Radio Access Network
NIST	National Institute of Standards and Technology
LAN	Local Area Network
SLA	Service Level Agreement
ONF	Open Networking Foundation
IRTF	Internet Research Task Force
SDNRG	Software Defined Networking Research Group
ODCA	Open Data Center Alliance

OS	Operating System
NOS	Network Operating System
VLAN	Virtual Local Area Network
WLAN	Wireless local Area Network
VAP	Virtual Access Point
D2D	Device to Device
ICN	Information-centric Networking
VMM	Virtual Machine Monitor
VM	Virtual Machine
NAT	Network Address Translation
DNS	Domain Name Service
VNF	Virtualized Network Function
VNFI	Virtualized Network Function Infrastructure
ETSI	European Telecommunications Standards Institute
ETSI ISGNFV	ETSI Industry Specification Group for Network Functions Virtualization
VNFFG	Virtualized Network Function Forwarding Graph
PoP	Point of Presence
VNFM	Virtualized Network Function Manager
VIM	Virtualized Infrastructure Manager
EMS	Element Management System
NSD	Network Service Descriptor
VNFD	Virtualized Network Function Descriptor
DB	Database
ICT	Information and Communications Technology
AC	Affective Computing
DBMS	Database Management System
FS	File System
DIP	Data Independence Principle
RDBMS	Relational Database Management System
SQL	Structured Query Language
2PC	Two-phase Commit

ACP	Atomic Commitment Protocol
RDF	Resource Description Framework
W3C	World Wide Web Consortium
MVCC/MCC	Multiversion Concurrency Control
UPnP	Universal Plug and Play
SSDP	Simple Service Discovery Protocol
OSGi	Open Service Gateway Initiative
RCA	Root Cause Analysis
CSAR	Cloud Service Archive
PD	Package Descriptor

Chapter 1

Introduction

This era is an undisputed witness of an exponential growth in the amount of traffic carried through mobile networks. According to the Cisco visual networking index, mobile data traffic has doubled during 2010–2011, while it is predicted that global mobile traffic will increase x1000 from 2010 to 2020 [1]. The surge in mobile traffic is primarily driven by the proliferation and the skyrocketed adoption of data-hungry mobile devices. In addition to this adoption, another paramount factor associated with the tremendous mobile traffic growth is the increasing demand for advanced multimedia services, such as Ultra-high Definition (UHD) quality of transmission and 3D video, as well as Augmented Reality (AR) and immersive experience. More specifically, mobile video accounts for more than 50% of global mobile data traffic, which is anticipated to rise to 73% by 2018 [1]. Additionally, social networking has become important for mobile users, introducing new consumption requirements and a huge amount of mobile data traffic. Apart from tremendous traffic growth, the escalating number of interconnected devices imposes challenges on the future mobile network. It is studied that everyone and everything will be interconnected under the umbrella of Internet of Everything (IoE), with tens to hundreds of devices belonging to every person in the future connected society.

The upcoming networking infrastructure and its support for Big Data will enable everything to be smart. The exploitation of data will be generated everywhere by both people and machines, and will be analysed in a real-time fashion to reveal useful information, from people's habits and preferences to the traffic condition on the streets, and health monitoring for patients and elderly people. With data constantly accumulated and the technologies of Big Data analytics rapidly developed, the great value hided behind data has gradually been revealed. Thus, it is highly important to exploit this precious evolution to improve the performance of mobile communications and maximize the revenue of operators. Traditional data analytics shows its inadequateness when encountered with the big cellular data. Firstly, traditional data analytics deals with structured data, while the large amount of app-based data is, however, generally unstructured. Secondly, the implementation of data analysis is traditionally confined within a department, or a business unit, and the final analytical conclusions come from very limited, local angles, rather than global perspectives. Third, the analytics mainly aims at transaction data, and pays less attention to the operational data, due to its incapability to make real-time decisions.

Big data analytics can extract much more insightful information than traditional data analytics. For example, the complete data related to a subscriber is usually fragmented in different business departments. Big data analytics is capable of collecting the scattered data to understand the user behavior and preferences from multiple perspectives to portray an integrated picture. Moreover, subscribers' living habits and timetables can be generally inferred from the usage of traffic over different time periods of a day.

From the network perspective, the evolution of Internet technologies has converged towards an IP packet-switched service, reformating the way of thinking, acting and living in the society. Nowadays, Internet delivers a enriched palette of services that include media entertainment (from audio and video to high-definition online games), personalisation (e.g., haptics, presence-based applications and location-based services) and more sensitive and safety-critical applications (e.g., e-commerce, e-Health, first responders, etc.). According to the International Telecommunication Union (ITU), the global Internet was being reached by more than 2.4 billion users around the world in June 2012, while going beyond [2]. Furthermore, the Cisco forecast on the usage of IP networks revealed that Internet traffic is evolving from a steadier to a more dynamic pattern. In this context, everything will be connected by the end of the decade, structuring the Internet of Things (IoT). A famous example is Machine to Machine (M2M) communications, exploiting sensor-based networking resulting in an additional driver for traffic growth. Thus, it turns out that the drivers of the future Internet are all kinds of services and applications, from low throughput rates (e.g., sensors and IoT data) to higher ones (e.g., high–definition video streaming), that need to be compatible to support various latencies and devices.

Nowadays, the prevalent networks are populated with a large and increasing variety of proprietary hardware appliances. The launch of a new Network Service (NS) often requires detecting the appropriate space and power. It is substantially difficult to achieve this and keep up with new technological trends and service innovations, as acceleration on making hardware lifecycles shorter than ever is present. Also, network infrastructures start to demand automated control capabilities for scalability, robustness and availability, especially in huge network environments, with the aim of reducing the impact of manual intervention which is becoming an expensive commodity [3]. Other concerns include increasing costs of energy, capital investment challenges and the problems imposed by design, integration and operation of increasingly complex hardware-based appliances. Consequently, the consideration of the two business expenses of Capital Expenditures (CAPEX) and Operating Expenses (OPEX) is of paramount importance. These growing limitations of the Internet, in terms of network management and best-effort forwarding, which have failed to meet the Quality of Service (QoS) and Quality of Experience (QoE) requirements for added-value applications, are well recognised in the research community, whether in academia or in industry.

Therefore, it is broadly proposed that the Internet architecture necessitates reformation and many proposals, including 'clean slate' approaches [4, 5]. It is evermore clear that a turning point is approaching in communication networks with a progressive introduction of Software Defined Networking (SDN) [6] and virtualisation of network functionalities to offer the required flexibility and reactivity [7, 8]. More specifically, SDN suggests the decoupling of the network control plane from the data plane, while Network Function Virtualisation (NFV) allows for instantiating many distinct logical network functions on top of a single shared physical network infrastructure. As recent research findings claimed, network resource over-provisioning can effectively achieve QoS differentiation in a scalable manner, whose approach is fundamental for the future Internet. While these technologies (i.e. SDN, Virtualisation and QoS over--provisioning) are promising to improve future networking performance, they are still in their infancy as further analysis and research are still deemed necessary.

The requirement for programmable networks have coupled SDN and Cloud Computing, which are broadly used together, at least for private clouds, but it probably will be over the next few years. Telecommunication Service Providers (TSPs) need the ability to deploy resources to companies' cloud in a rapid and transparent way, as also to have network visibility across the entire network to handle problems, traffic bottlenecks in real time, etc. SDN can provide the needed omniscience and do so in a secure manner. In addition, with hundreds or thousands of customers, the need to deploy virtual switches and routers to link a customer's equipment throughout and between data centers requires quick provisioning and centralized command and control. From the company's side, there are no service provider needs, but still the virtual deployment is desirable (such as firewalls or load balancers) to their customers, namely the end-users, developers, email and database administrators, etc. As the necessities become larger and the computing resources for a single use case spread out across one or more data centers, the ability to deploy and manage all of the equipment becomes more challenging. If you then add the needs for security (such as Intrusion Detection or Prevention Systems (IDS/IPS), firewalls, and antivirus deployments) and reporting, the complexity and chances for making a mistake increase tremendously. Further complicating the scenario is the necessity of load balancing as projects bloom and consume more resources. Consequently, the need for rapid scalability and provisioning of devices becomes more prevalent. The need for automated provisioning of virtual networking equipment almost becomes a requirement, and with all the new virtual devices, centralized command and control becomes a must, defining cloud computing, SDN, and NFV as the optimal solutions.

1.0.1 Motivation and Research Challenges

In the SDN/NFV ecosystem, network services are provided as single Virtual Network Functions or chains of them, each instantiated and executed on diverse or dedicated servers. As personalized on-demand services, traffic volume and competitive pressures increase, Telecommunication Service Providers (TSPs) must strike a balance between minimizing expenses and maximizing their market power. SDN/NFV technologies have begun to provide ground for opportunities to this challenge. Consequently, the next critical step is to fully commercialize virtualized services and render them operationally ready to perform at scale with a large variety of partners. Network and service virtualization have become more than concepts, as they are business imperatives that necessitate effective transformation methods, proven technology and solutions that are ready for the rigors of a dynamic marketplace.

The aim of this MSc thesis is to develop a scalable and efficient NFV marketplace system, where third-party VNF providers sell VNF-as-a-service (VNFaaS), adapt their pricing policies according to network dynamics and help TSPs make the transition to virtualized networks and services. Designed to combat the main key challenges of storage, commercialization, multivendor services and optimal efficiency in the NFV landscape, the NFV Marketplace aims beyond a plain data repository to the delivery of novel added-value services through the exploitation and the mining of the available information. Through the aggregation of the stored information, the NFV Marketplace triggers the design of recommender systems and fulfil the need for filtering, prioritizing and efficiently deliver information in such way that supports the decision making process of the users. Thus, our solution offers a decision support mechanism, which aims at introducing recommendations to the end users and supporting the infrastructure owner/operator regarding the selection and use of assets in the NFV landscape.

1.0.2 Thesis Contribution

The contributions of this MSc dissertation move towards to the development of a novel enabling framework, the *NFV Marketplace*, allowing the diverse stakeholders the key storage component not only to deploy Network Services, but also to offer them to their customers added-value functionalities. This holistic framework supports the lifecycle management and storage of network services, as well as data analytics algorithms as novel services. In more detail:

Chapter 2: Fundamentals of 5G Networks

The second chapter presents an introduction to the topic of 5G Networks and its relation with the principles of SDN and NFV. In more detail, it describes the proposed architectures, along with core elements of these principles.

Chapter 3: Review of NoSQL Databases

In the third chapter, a general review of the NoSQL databases is provided. More specifically, a variety of core properties from SQL databases are presented with the connection to the NoSQL ecosystem. Additionally, the NoSQL databases are meticulously studied, accompanied by their properties and concepts.

Chapter 4: The NFV Marketplace

In this chapter, the design of the open *NFV marketplace* is meticulously presented, going beyond the plain NFV data repository to an intelligent trading agent of cloud services. More specifically, the core functionalities and added-value algorithms of the NFV Marketplace are analyzed, accompanied by several numerical results which are extracted from experiments on an actual 5G ecosystem.

Chapter 2

Fundamentals of 5G Networks

5G wireless technology is paving the way to change the future ubiquitous and pervasive networking, wireless applications, and user QoS/QoE, centering its design objectives around efficiency, scalability and versatility. To sustain their commercial viability, 5G networks must be significantly efficient in terms of energy and resource management. Connecting a massive number of terminals and battery operated devices necessitates the development of scalable and versatile network functions that cope with a wider range of service requirements, including low power machine-type communication, high–data rate multimedia, and delay-sensitive applications, among many other services. The efficiency, scalability, and versatility objectives of 5G direct the 5G community toward finding innovative but simple implementations of 5G network functions. Undisputedly, the rapid evolution of mobile communications delivers a remarkable impact on the social and economic development. The design of the 5G mobile wireless standard introduces the reconstruction of the prevalent network infrastructure, the enhancement of the wireless connectivity and assurance of QoS/QoE levels, by delivering the support of a plethora of innovating services. In order to bridge the gap between the capacity improvement and the scalable connectivity, the advent of 5G mobile networks will introduce paramount changes from the perspective of flexible network management [9]. The TSPs dictate requirements such as high throughput, high reliability, low latency, increased capacity, availability and connectivity, and dynamic bandwidth allocation from the 5G cloud Radio Access Networks (RAN). The principles of SDN and NFV can effectively diminish these challenges with aligned optimization of short-service life-cycles. Consequently, the digital transformation of network infrastructure through NFV/SDN is anticipated to play a pivotal role with the respect to the commercialization of 5G network [10]. Through the virtualization and the abstraction of lower level functions, SDN and NFV provide the necessary tools for the composition of the 5G architecture. The optimum design and management in these ecosystems leverages the peak benefits of NFV to reduce the CAPEX/OPEX in 5G networks by means of dynamic resource allocation and traffic load balancing [11].

In this chapter, the core of the Cloud Computing concept, along with the SDN and NFV concepts are introduced, focusing on the aspects that are closely related to the virtualization of the 5G infrastructure. The exploitation of these concepts in the 5G networks offers agility enhancement, exploitation and maintenance automation compared to the legacy technologies, while it provides the necessary tools to simplify the vertical system organization [12, 13].

2.1 Cloud Computing

The widely accepted definition of Cloud Computing was delivered by the National Institute of Standards and Technology (NIST) in 2011, concerning the wide feedback [14]:

Definition 2.1 *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*

Much of the Internet is dedicated to the access of content-based IT resources published via the World Wide Web. On the other hand, IT resources provided by cloud environments are dedicated to supplying back-end processing capabilities and user-based access to these capabilities. Another key distinction is that it is not necessary for clouds to be Web-based even if they are commonly based on Internet protocols and technologies. Cloud protocols refer to standards and methods that allow computers to communicate with each other in a pre-defined and structured manner. A cloud can be based on the use of any protocols that allow for the remote access to its IT resources. But there's more going on under the hood than to simply equate cloud computing to the Internet. In essence, Cloud Computing is a construct allowing the access to applications that actually reside at a location other than a plain computer or other Internet-connected device while most often, this will be a distant datacenter. The main benefit of Cloud Computing is that another company hosts application of the customers (or suite of applications, for that matter), handling the costs of servers and managing the software updates. Additionally, there is no cost of purchasing hardware components, which will result in fewer CAPEX/OPEX. Consequently, the definition of Cloud Computing model is composed of five essential characteristics, three service models, and four deployment models. The five essential characteristics of the Cloud Computing are listed below [14]:

- **On-demand self-service**: A consumer can receive unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.

- **Broad network access**: Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).

- **Resource pooling**: The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g. country, state, or data center). Examples of resources include storage, processing, memory, and network bandwidth.

- **Rapid elasticity**: Capabilities be can elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.
- **Measured service**: Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of the service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled and reported, providing transparency for both the provider and consumer of the utilized service.

2.1.1 Service and Deployment Models

The three defined service models of Cloud Computing model are the following [14]:

- **Cloud Infrastructure as a Service (IaaS)**: The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, deployed applications, and possibly limited control of select networking components (e.g., host firewalls).
- **Cloud Platform as a Service (PaaS)**: The capability provided to the consumer is to deploy onto the cloud infrastructure consumer–created or acquired applications–created using programming languages and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly application hosting environment configurations.
- **Cloud Software as a Service (SaaS)**: The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through a thin client interface such as a web browser (e.g., web–based email). The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user–specific application configuration settings.

On the same definition, four deployment models are defined as [14]:

- **Private cloud**: The cloud infrastructure is operated solely for an organization. It may be managed by the organization or a third party and may exist on premise or off premise.
- **Community cloud**: The cloud infrastructure is shared by several organizations and supports a specific community that has shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be managed by the organizations or a third party and may exist on premise or off premise.
- **Public cloud**: The cloud infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services.

- **Hybrid cloud**: The cloud infrastructure is a composition of two or more clouds (private, community, or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load–balancing between clouds).

2.1.2 Resources and Scalability

In a simple, topological sense, a cloud computing solution is made up of *resources*, physical or virtual IT-related artifact that can be either software-based, such as a virtual server or a custom software program, or hardware-based, such as a physical server or a network device. More specifically, the discrete components that are utilized are the *clients*, the *data center*, and *distributed servers*:

- **Clients**: As in the plain Local Area Network (LAN), clients are the devices that the end-users interact with to manage their information on the cloud.
- **Data center**: The data center is the collection of servers where the application to which you subscribe is housed and accessed through the Internet. A growing trend in the IT world is virtualizing servers, which is essentially software that can be installed allowing multiple instances of virtual servers to be used.
- **Distributed Servers**: Often, servers are in geographically disparate locations, giving the service provider more flexibility in options and security.

The party that provides cloud-based IT resources is called the *cloud provider*, while the party that uses cloud-based IT resources is the *cloud consumer*. These terms represent roles are usually assumed by organizations, in relation to clouds, and corresponding cloud provisioning contracts. Although a cloud is a remotely accessible environment, not all IT resources residing within a cloud can be made available for remote access. For example, a database or a physical server deployed within a cloud may only be accessible by other IT resources that are within the same cloud. A software program with a published Application Programming Interface (API) may be deployed specifically to enable access by remote clients. Additionally, a *cloud service* is any IT resource that is made remotely accessible via a cloud. Unlike other IT fields that fall under the service technology umbrella, such as service-oriented architecture, the term ”service” within the context of cloud computing is especially broad. A cloud service can exist as a simple Web-based software program with a technical interface invoked via the use of a messaging protocol, or as a remote access point for administrative tools or larger environments and other IT resources. The driving motivation behind cloud computing is to provide IT resources as services that encapsulate other IT resources, while offering functions for clients to use and leverage remotely.

A multitude of models for generic types of cloud services have emerged, most of which are labelled with the “as-a-service” suffix. The cloud service consumer is a temporary runtime role for the client assumed by a software program when it accesses a cloud service. Common types of cloud service consumers can include software programs and services capable of remotely accessing cloud services with published service contracts, as well as workstations, laptops and mobile devices running software capable of remotely accessing other IT resources positioned as cloud services. Cloud service usage conditions are typically expressed in a Service Level Agreement (SLA) that is the human-readable part of a service contract between a cloud provider and cloud consumer that describes QoS features, behaviours, and limitations of a cloud-based service or other provisions. An SLA provides details of various measurable characteristics related to IT outcomes, such as uptime, security

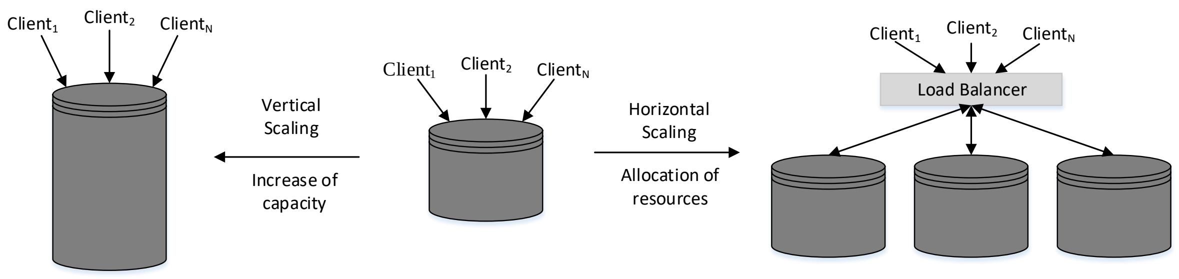


Figure 2.1: *Vertical and horizontal scaling*

Table 2.1: *Comparison of horizontal and vertical scaling schemes*

Horizontal Scaling	**Vertical Scaling**
Reduced costs via commodity hardware components	Increased cost via specialized servers
Rapid availability of IT resources	Rapid availability of IT resources
Resource replication and automated scaling	Additional setup is normally needed
Necessity of additional IT resources	No necessity of additional IT resources
No limitation of hardware capacity	Limitation of hardware capacity

characteristics, and other specific QoS features, including availability, reliability, and performance. Since the implementation of a service is hidden from the cloud consumer, an SLA becomes a critical specification.

The manipulation of resources resides through the feature of *scalability*. From an IT resource perspective, scaling represents the ability of the IT resource to handle increased or decreased usage demands and represents one of the most valuable and predominant feature of cloud computing. Scalability enables the accommodation of larger workloads without disruption or complete transformation of existing infrastructure. The main types of scaling are the following:

- **Horizontal Scaling**: The allocating or releasing of IT resources that are of the same type is referred to as horizontal scaling, as depicted in Fig.2.1. The horizontal allocation of resources is referred to as scaling out and the horizontal releasing of resources is referred to as scaling in. Horizontal scaling is a common form of scaling within cloud environments.

- **Vertical Scaling**: When an existing IT resource is replaced by another with higher or lower capacity, vertical scaling is considered to have occurred, as depicted in Fig.2.1. Specifically, the replacing of an IT resource with another that has a higher capacity is referred to as scaling up and the replacing an IT resource with another that has a lower capacity is considered scaling down. Vertical scaling is less common in cloud environments due to the downtime required while the replacement is taking place, as also listen in Table 2.1.

2.2 Foundations of the SDN Approach

The promising SDN architecture provides programmability, flexibility and reliability over networks. Network operators can implement custom protocols and rules with common programming languages and achieve flexible control management over network services, such as routing, traffic engineering, QoS and security. Network can easily amend itself with main criterion the users' requirements as network management and configurations can be automated through the centralized controller and standard open API, making the network scale and reformate easily [15].

The notion of programmable networks has been conceived many years ago. SOFTNET [16], Active Networking [17], OPENSIG [18] and GSMP [19], IEEE P1520 Standards Initiative [20], 4D Project [21] and Soft-Router [22] and finally NETCONF [23], Ethane [24] and SANE [25] architectures were sequentially proposed to answer the idea of programming network among the years. SANE was developed as a prototype, considering a logical server that performs all access control decision. Ethane separated controller and Ethane switch toward providing policy and security by an identity-oriented access. Yet, in 2010 according to RFC5810 by IETF, ForCES (Forwarding and Control Element Separation) [26] was announced, standardizing the communication between controller and network elements. The Open Networking Foundation (ONF), a user-led organization dedicated to promotion and adoption of SDN, manages the *OpenFlow* standard. ONF defines OpenFlow as the first standard communications interface defined between the control and forwarding layers of an SDN architecture. OpenFlow allows direct access to the forwarding plane of network devices, such as switches and routers in both physical and virtual level, and their consequent manipulation. The difference between ForCES and OpenFlow lies in their network architectures. ForCES assumed no variance in the network architecture, where controller and network elements are situated in one single device, as regards to this point that they are apart from each other, while OpenFlow separates the controller and network elements physically. Several efforts in the field of SDN have been taken until now. Internet Research Task Force (IRTF), defined the Software Defined Networking Research Group (SDNRG), intended to figure out future SDN approaches and challenges.

2.2.1 Key Principles and Infrastructure Requirements of SDN

As SDN draw the broad attention, ONF described its view with respect to the extent of the SDN Architecture to all interested parties, as also it supplies guidance to individual ONF Projects, Discussion Groups and Working Groups with respect to the scope of SDN, the preferred architectural approach, and key requirements. Detailed specification of architectures, technologies and standards (e.g., individual behaviour and interface specifications) is the purview of specific working groups. Though, the key principles of SDN are listed below:

1. SDN enables the programmatic and abstracted interaction with the network. The necessary classes of interaction include control, provisioning, configuration, management, and/or monitoring.
2. The network control function is decoupled from the controlled network elements while being logically centralized. The control function and selected management/monitoring functions are housed in an *SDN Controller*.
3. The *SDN Controller* must be able to exert programmatic direct control of forwarding behaviour, and actually define the network (e.g., its logical topology), not merely influencing/configuring it.

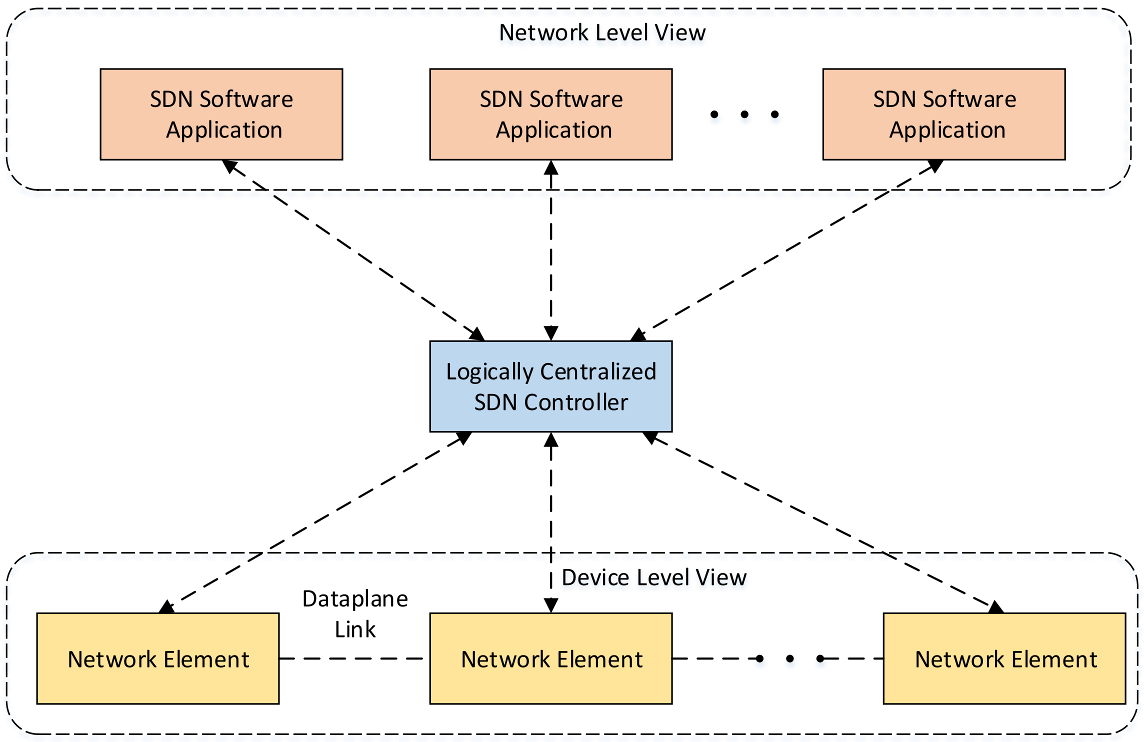


Figure 2.2: *The SDN enablement of software-oriented and abstracted interaction in the network*

Fig.2.2 depicts a brief visualization of the aforementioned characteristics. Any architecture that is compliant and meets the requirements elucidated in the specification is accepted as meeting the embodied definition of SDN, whether or not the technology is standardized by ONF or others. Since SDN comprise a powerful solution to the leveraging of the network capabilities, the Open Data Center Alliance (ODCA) provides a useful and concise list of requirements [27]:

1. **Model management**: Allowance of network management software to operate at a model level, rather than implementing conceptual changes by reconfiguring individual network elements.
2. **Mobility**: Accommodation of mobility through the mobile user devices and virtual servers with control functionality.
3. **Maintainability**: Introduction of new features and capabilities (software upgrades, patches) must be seamless with minimal disruption of operations.
4. **Automation**: Policy changes must be automatically propagated with the view of reducing manual work and errors.
5. **Integrated security**: Integration of security in network applications seamlessly as a core service instead of as an add-on solution.
6. **On-demand scaling**: Feasibility of scaling up/down the network and services to support on-demand requests.

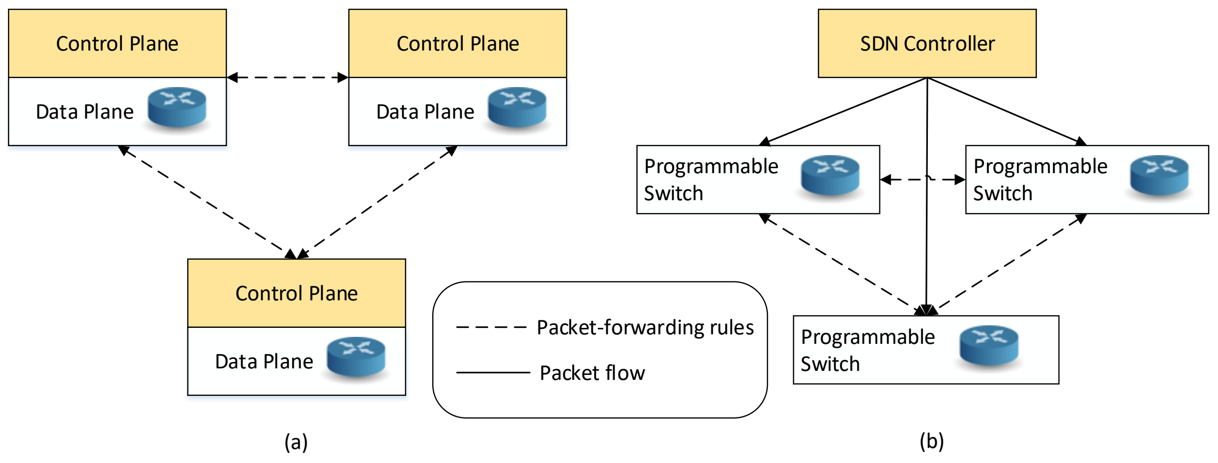


Figure 2.3: *Control and data planes for (a) the traditional network architecture and (b) the SDN approach*

7. **Adaptability**: Adjusting networks dynamically, based on application needs, business policy, and network conditions.

2.2.2 The SDN Architecture

An analogy can be drawn between the way in which computing evolved from closed, vertically integrated and proprietary systems into an open approach to computing and the evolution coming with SDN. In the early decades of computing, various vendors, such as IBM and DEC, provided an integrated product, with a proprietary processor hardware, unique assembly language and Operating System (OS). In this environment, large customers tended to be locked into one vendor, dependent primarily on the applications offered by this specific vendor. Migration to another vendor's hardware platform resulted in major upheaval at the application. Thus, the networking environment introduced several limitations, also prevalent in the pre-open era of computing. Rather than designing interoperable software solutions, the difficulty lies in the lack of integration between applications and network infrastructure. Traditional network schemes are inadequate to meet the necessities of the growing volume and variety of traffic.

The central notion behind SDN is the feasibility of developers and network managers to have the same type of control over the common network equipment. The SDN approach splits the switching function of a switch between a data plane and a control plane that are on separate devices, as depicted in Fig.2.3. The data plane is simply responsible for the forwarding of the packets, while the control plane provides the intelligence in the design of the routes, setting priority and routing policy parameters to meet QoS requirements and cope with the shifting traffic patterns. Open interfaces are defined so that the switching hardware to present a uniform interface, regardless of the details of internal implementation. Similarly, open interfaces are defined to enable networking applications to communicate with the SDN controllers. The two main entities involved in forwarding packets through routers are the *control function*, responsible for the route of the traffic, and the *data function*, forwarding data, based on control-function policy. Prior to SDN, these functions were performed in an integrated fashion at each network device (router, bridge, packet switch, etc). Control in such a traditional network is exercised by means of a routing and control network protocol that is implemented in each network

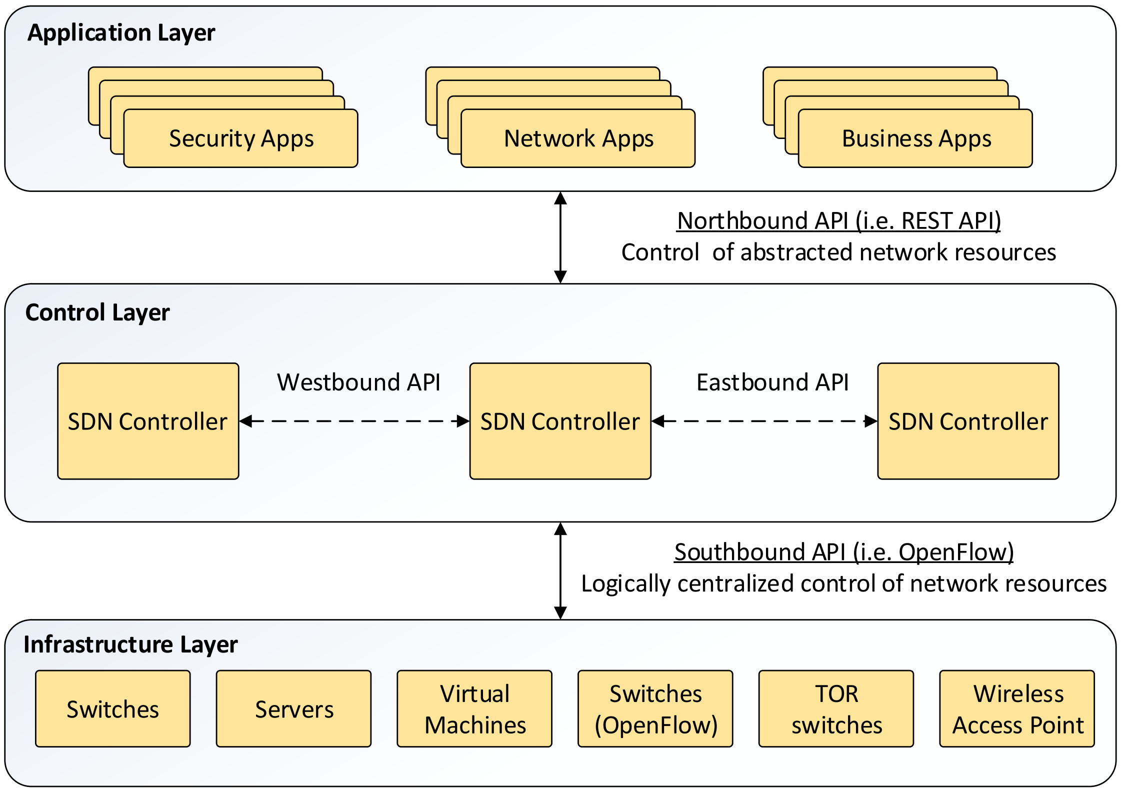


Figure 2.4: *Software Defined Network Architecture*

node.

As this is relatively inflexible and requires all the network nodes to implement the same protocols, SDN delivers novel advantages through a central SDN controller that performs all complex functionalities, such as routing, naming, policy declaration, and security checks. The set of the deployed SDN controllers constitutes the SDN *control layer* as depicted in Fig.2.4. The SDN controller defines the data flows that occur in the SDN *infrastructure layer*, including the physical switches and virtual switches. Each flow through the network is configured by the controller, verifying that the communication is permissible by the network policy. If the controller allows a flow requested by an end system, it computes a route for the flow to take and adds an entry for that flow in each of the switches along the path. With all complex function subsumed by the controller, switches simply manage flow tables whose entries can only be populated by the controller. In both cases, the switches are responsible for forwarding packets. The internal implementation of buffers, priority parameters, and other data structures related to forwarding can be vendor-dependent. However, each switch must implement a model, or abstraction, of packet forwarding that is uniform and open to the SDN controllers. This model is defined between the control and the infrastructure layer, in terms of an open API named as the *southbound API*. The most prominent example of such an open API is OpenFlow, with its specification defining both a protocol between the control and data planes and an API by which the control layer can invoke the OpenFlow protocol.

2.2.3 Infrastructure Layer

An SDN infrastructure is composed of a set of networking equipment, from switches up to routers and middlebox appliances. The main difference with a typical network resides in the fact that those traditional physical devices are now simple forwarding elements without embedded control or software functionalities to take autonomous decisions. The network intelligence is subtracted from the data plane of the devices to a logically-centralized control system, i.e., the network operating system and applications, as depicted in the Fig.2.3. More importantly, as these new networks are built on top of open and standard interfaces, a crucial approach for ensuring configuration, communication compatibility and interoperability among the different data and control plane of the devices is necessary. In other words, these open interfaces enable controller entities to dynamically program heterogeneous forwarding devices, which is something difficult in traditional networks, due to the large variety of proprietary and closed interfaces and the distributed nature of the control layer. In an SDN architecture, there are two main elements, the *controllers* and the *forwarding devices*, depicted in Fig.2.4. An infrastructure layer device is a hardware or software element specialized in packet forwarding, while a controller is a software stack/”brain”, running on a commodity hardware platform. An OpenFlow-enabled forwarding device is based on a pipeline of flow tables where each entry of a flow table has three parts, namely the *matching rule*, the *actions to be executed on matching packets*, and the *counters keeping statistics of matching packets*. This high-level and simplified model is derived from the OpenFlow and is currently the most widespread design of SDN infrastructure layer devices. Nevertheless, other specifications of SDN-enabled forwarding devices are also being pursued [28, 29]. Inside an OpenFlow device, a path through a sequence of flow tables defines how packets should be handled. When the arrival of a new packet is enabled, the lookup process starts in the first table and ends either with a match in one of the tables of the pipeline or with a miss. A flow rule can be defined by combining different matching fields. If there is no default rule, the packet will be discarded. However, the common case is to install a default rule which implies that the switch will send the packet to the controller. The priority of the rules follows the natural sequence number of the tables and the row order in a flow table.

The bridge of control and forwarding elements is feasible through the *Southbound Interfaces/APIs*, which are crucial instruments for separating control and infrastructure layer functionalities. However, these APIs are still strictly tied to the forwarding elements of the underlying physical or virtual infrastructure. As a central component of its design, the southbound APIs represent one of the major barriers for the introduction and acceptance of any new networking technology. In this light, the emergence of SDN southbound API proposals, such as OpenFlow [30], is faced as welcome from the industry. These standards promote interoperability by allowing the deployment of vendor-agnostic network devices. This has already been demonstrated by the interoperability presented between OpenFlow-enabled equipments from different vendors. OpenFlow is the most widely accepted and deployed open southbound standard for the SDN approach, providing a common specification to implement OpenFlow-enabled forwarding devices, and for the communication channel between data and control plane devices (e.g., switches and controllers). The OpenFlow protocol provides three information sources for network operating systems, described in the later sections. On the first level, event-based messages are sent by forwarding devices to the controller as soon as a link or port change is triggered. Secondly, an amount of flow statistics are generated by the forwarding devices, collected by the controller. Thirdly, packet-in messages are sent when there is no knowledge what to do with a new incoming flow or because there is an

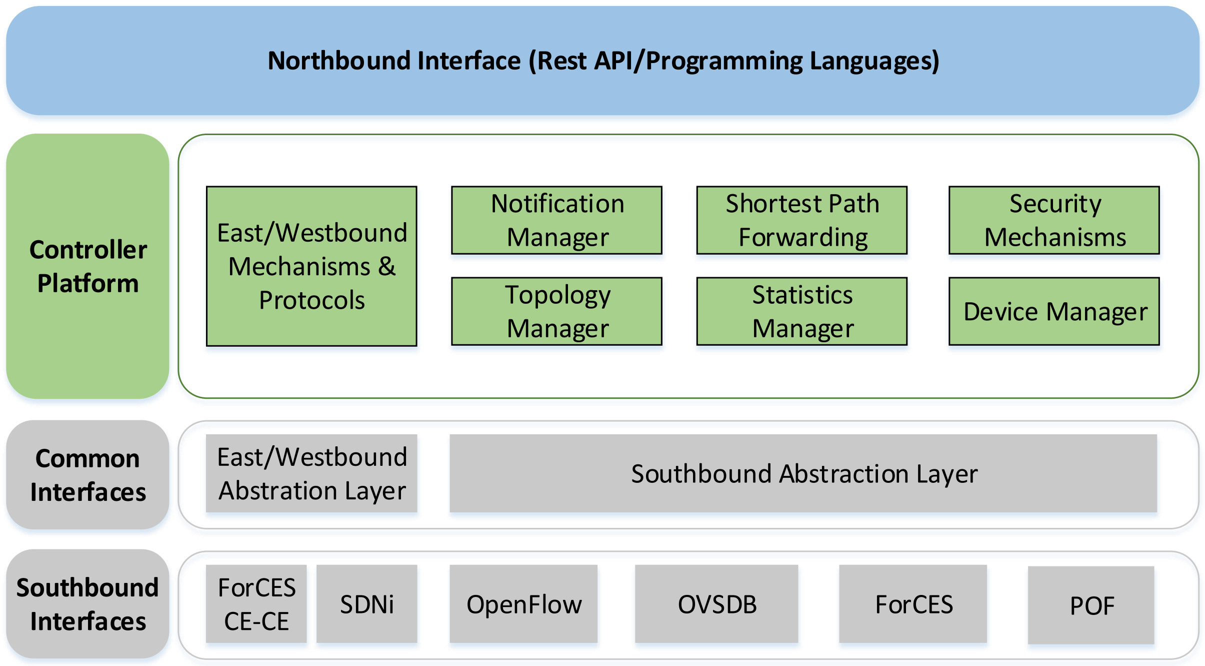


Figure 2.5: *Elements and interfaces of an SDN Controller*

explicit "send to controller" action in the matched entry of the flow table. These information channels are the essential means to provide flow level information to the network operating system. Eventually in general, apart from the OpenFlow, there are several southbound interfaces for SDN [26, 28, 31–37].

2.2.4 Control Layer

The SDN control layer is responsible for the mapping of the application layer service requests into specific actions and directives to the infrastructure layer switches and supplies applications with information about infrastructure plane topology and activity. The control layer is implemented as a server or cooperating set of servers, known as *SDN controllers*. A simplified structure of the available elements of an SDN controller is depicted in Fig.2.5. The facilitation of network management and delivery of solutions to networking problems is offered by a *Network Operating System* (NOS), by means of the logically-centralized control, along with generic functionalities such as network state and network topology information, device discovery, distribution of network configuration, etc. On this basis, the definition of network policies becomes independent from the low-level details of data distribution among routing elements. Such software can arguably create a new environment, capable of alleviating innovation at a faster pace by easing the issue of creating new network protocols and network applications. Consequently, the NOS is a critical element in an SDN architecture as it is the main supporting-piece for the control logic (applications) to generate the network configuration based on the defined policies. Similar to a traditional operating system, the control platform abstracts the lower-level details of connecting and interacting with forwarding devices. The base network service functions are included in the *Controller Platform* level, where the essential functionality should be provided. The included functionalities are base services of operating systems, such as program execution, I/O operations control, communications, protection, etc, while being used by other operating system level services and user applications. Similarly, functions are essential network control functionalities that network applications may use in building its logic. As depicted in Fig.2.5, the following functions are included:

- **Shortest path forwarding**: Exploitation of routing information of switches with the aim of establishing preferred routes
- **Notification Manager**: Receival, process and forwarding of application events, such as security alarms, state changes, etc
- **Security mechanisms**: Isolation and security provision between applications and services
- **Statistics manager**: Aggregation of traffic data of switches
- **Device manager**: Configuration of switch parameters/attributes and management of route table
- **Topology manager**: Build and maintenance of data topology

Among the available set of SDN controllers and control platforms [38–44], two categories can be differentiated from an architectural point of view, namely the *distributed* and the *centralized* SDN controllers. The former category represents a scalable solution to meet the requirements of potentially any environment, from small to large-scale networks. A distributed SDN controller can be a centralized cluster of nodes or a physically distributed set of elements. While the centralized solution can offer high throughput for very dense data centers, the physically distributed set of elements can be more resilient to different kinds of logical and physical failures. A cloud provider that incorporates several servers interconnected by a wide area network may require a hybrid approach, with clusters of controllers inside each data center and distributed controller nodes in the different sites. The distributed SDN controllers present the problem of consistency with a portion offering weak consistency semantics, which means that data updates on distinct nodes will eventually be updated on all controller nodes. Other solutions offer strong consistency, ensuring that all controller nodes will read the most updated property value after a write operation. Similarly, another common property is fault tolerance, which denotes the dependency in server failure to other servers. So far, despite some controllers tolerating crash failures, they do not tolerate arbitrary failures, which means that any node with an abnormal behaviour will not be replaced by a potentially well behaved one. On the other hand, a centralized SDN controller comprises an entity, responsible for all the forwarding devices of the network. Thus, it represents a single point of failure and has scaling limitations. The design strategy of centralized controllers aims either at delivering high concurrency, to achieve the desired throughput required by enterprise class networks and data centers through multi-threaded designs, or at specific environments such as data centers, and cloud infrastructures.

In the case of distributed SDN controllers, special cases of interfaces are deemed necessary for the inner-communication of the distributed SDN controllers. *East/westbound APIs* are special middleware components incorporated for the support of distributed controllers, where each controller implements its own east/westbound API. The functionalities of these interfaces include import/export of data between controllers, algorithms for data consistency models and monitoring/notification capabilities (e.g., check if a controller is up or notify a take over on a set of forwarding devices). The identification and provision of common compatibility and interoperability between different controllers is substantially necessary with the aim of standardization in east/westbound interfaces. More specifically, the famous protocol of SDNi defines common requirements to coordinate flow setup and exchange reachability information across multiple domains [45]. Such protocols can be used in an orchestrated and interoperable way to provide more scalable and dependable distributed control platforms. Interoperability can be leveraged to increase the diversity of the control platform element, which increases the system robustness through the reduction of the probability of common faults, such as software faults.

The Northbound and Southbound interfaces are two main key abstractions of the SDN ecosystem. Regardless of the fact that southbound interface has already the well-established proposal of OpenFlow, a common Northbound Interface is not yet discovered [46]. It is to be expected a specification of a Northbound Interface to arise as the evolution of SDN arises even more. An abstraction that would allow network applications not to depend on specific implementations is important to explore the full potential of SDN [46–55]. The Northbound Interface is mostly a software ecosystem, being differentiated from the functionalities of the hardware one as is the case of the southbound APIs. In this environment, the implementation is commonly the forefront driver, while standards emerge later and are essentially driven by wide adoption. Nevertheless, an initial and minimal standard for Northbound Interfaces can still play an important role for the future of SDN. Thus, a common consensus is that Northbound APIs are indeed important but that it is indeed too early to define a single standard right now. The experience from the development of different controllers will certainly be the basis for coming up with a common application level interface. Among the proposed schemes of northbound APIs, it is unlikely that a single Northbound Interface will emerge as the winner, as the requirements for different network applications are quite different. APIs for security applications are likely to be different from those for routing or financial applications. One possible path of evolution for Northbound APIs are vertically-oriented proposals, before any type of standardization occurs, a challenge the ONF has started to take action in parallel to open-source SDN developments. The ONF architectural work includes the possibility of Northbound APIs providing resources to enable dynamic and granular control of the network resources from customer applications, eventually across different business and organizational boundaries.

2.2.5 Application Layer

The application layer is the set of applications that leverage the functions offered by the Northbound Interface to implement network control and operation logic, including applications such as routing, firewalls, load balancers, monitoring, and so forth. Essentially, a management application defines the policies, which are ultimately translated to southbound-specific instructions that program the behaviour of the forwarding devices.

2.2.5.1 Programming Languages

The power of the SDN approach to networking lies in the support in providing network applications to monitor and manage network behaviour. The SDN control layer provides the functions and services that facilitate rapid development and deployment of network applications. While the SDN infrastructure and control layer are well defined, there is much less agreement on the nature and scope of the application layer. At minimum, the application plane includes a number of network applications, specifically dealing with network management and control. There is no agreed-upon set of such applications or even categories of such applications.

Further, the application layer may include general-purpose network abstraction tools and services that might also be viewed as part of the functionality of the control plane. The application layer is faced as the incorporation of the control logic, translated into commands to be installed in the infrastructure layer, dictating the behaviour of the forwarding devices. The application layer contains applications and services that define monitor, and control network resources and behaviour. These applications interact with the SDN control layer via application-control interfaces, so as for the SDN control layer to automatically customize the behaviour and the properties of network resources. The programming of an SDN application makes use of the abstracted view of network resources provided by the SDN control layer by means of information and data models exposed via the application-control interface. For example, a simple routing application follows a logic of a path definition through which packets will flow from a point A to a point B. For this purpose, a routing application has to decide on the path to use, based on the topology input, and instruct the controller to install the respective forwarding rules in all forwarding devices on the chosen path, from A to B.

Two essential attributes of virtualization solutions are the capability of expressing modularity and of allowing different levels of abstractions while still guaranteeing desired properties, such as protection. For instance, virtualization techniques can allow different perspectives of a single physical infrastructure. As an example, one virtual switch could be represented as a combination of several underlying forwarding devices. This intrinsically simplifies the task of application developers as they are not obliged to think about the sequence of switches, with forwarding rules to be installed, but rather see the network as a simple switch. Such kind of abstraction significantly simplify the development and deployment of complex network applications, such as advanced security related services. Programming languages have been proliferating for decades. Both academia and industry have evolved from low-level hardware-specific machine languages, such as assembly for x86 architectures, to high-level and powerful programming languages, such as Java and Python. The advancements of portable and reusable code has driven a significant shift on the computer industry [56].

In a similar view, programmability in networks is starting to move from low-level machine languages, such as OpenFlow, to high-level programming languages [57–62]. Assembly-like machine languages, such as OpenFlow and POF, essentially adopt the behavior of forwarding devices, forcing developers to focus on low-level details rather than on the problem itself. Raw OpenFlow programs have to deal with hardware behaviour details such as overlapping rules, the priority ordering of rules, and data-plane inconsistencies that arise from in-flight packets with flow rules under installation [63]. The usage of these low-level languages confound the reuse software, to create extensive code, and leads to a more error-prone development process [64, 65]. Though, several challenges exist in programming languages in SDNs. A first example is in pure OpenFlow-based SDNs, where it is hard to ensure that multiple tasks of a single application (e.g., routing, monitoring, access control) is entirely independent from each other as rules generated for one task should not override the functionality of another task. Another example is the execution of multiple applications on a single controller. Typically, each application generates rules based on its own needs and policies without further knowledge about the rules generated by other applications. As a consequence, conflicting rules can be generated and installed in forwarding devices, creating problems for network operation. Programming languages and runtime systems can help to solve these problems that would be otherwise hard to prevent.

One of the main roles of programming language abstractions is the provision of the capability of creating and writing programs for virtual network topologies. This concept is similar to object-oriented programming, where objects abstract both data and specific functions for application developers, making it easier to focus on

Table 2.2: Programming languages in the SDN application layer

Name	Programming Model	Comments
Frenetic [58]	Functional	Design to surpass race conditions through well-defined high-level programming abstractions
Maple [66]	Functional	Provision of highly-efficient multi-core scheduler
Merlin [67]	Logic	Mechanisms for delegating management of sub-policies to tenants without violating global constraints
nlog [68]	Functional	Mechanisms for data log queries over a number of tables and production of immutable tuples for reliable detection and propagation of updates
NetCore [60]	Functional	High level programming language providing means for expressing packet-forwarding policies
Procera [61]	Functional/Reactive	Inclusion of set of high-level abstractions to alleviate the description of reactive and temporal behaviors
Pyretic [62]	Imperative	Ability of defining network policies at a high level of abstraction, offering transparent composition and topology mapping
FML [57]	Dataflow/Reactive	High-level policy description language
Flog [65]	Logic/Event-driven	Combination of FML and Frenetic, providing an event-driven and forward-chaining logic programming language
Nettle [59]	Functional/Reactive	Based on streams instead of events
NetKAT [69]	Functional	Designed on Kleene algebra for reasoning about network structure
FatTire [70]	Functional	Enable of regular expressions to describe network paths and respective fault tolerance requirements

solving a particular problem without worrying about data structures and their management. For instance, in an SDN context, instead of generating and installing rules in each forwarding device, one can think of creating simplified virtual network topologies that represent the entire network, or a subset of it. The programming languages or runtime systems should be responsible for generating and installing the lower-level instructions required at each forwarding device to enforce the user policy across the network. With such kind of abstractions, developing a routing application becomes a straightforward process. Similarly, a single physical switch could be represented as a set of virtual switches, each of them belonging to a different virtual network. These two examples of abstract network topologies would be much harder to implement with low-level instruction sets. In contrast, a programming language or runtime system can more easily provide abstractions for virtual network topologies.

Several programming languages have been proposed for SDNs, many of these listed in the Table 2.2. The great majority of these proposals incorporate abstractions for OpenFlow-enabled networks. The predominant programming paradigm is the declarative one, with a single exception, *Pyretic*, which is an imperative language. Most declarative languages are functional, while but there are instances of the logic and reactive types. The purpose and the expressiveness power vary in the programming choices, while the end goal is almost common as to provide higher-level abstractions to facilitate the development of network control logic. Programming languages such as FML, Nettle and Procera are functional and reactive. Policies and applications written in these languages are based on reactive actions triggered by events. Such languages allow users to declaratively express different network configuration rules, such as access control lists (ACLs), virtual LANs (VLANs), and many others. Rules are essentially expressed as allow-or-deny policies, which are applied to the forwarding elements to ensure the desired network behaviour. Other SDN programming languages such as Frenetic, Hierarchical Flow Tables (HFT), NetCore and Pyretic were designed with the simultaneous goal of efficiently expressing packet-forwarding policies and dealing with overlapping rules of different applications, offering advanced operators for parallel and sequential composition of software modules. Additionally, FatTire is an example of a declarative language that heavily relies on regular expressions to allow programmers to describe network paths with fault tolerance requirements. For instance, each flow can have its own alternative paths for dealing with failure of the primary paths. Interestingly, this feature is provided in a very programmer-friendly way, with the application programmer having only to use regular expressions with special characters, such as an asterisk. In the particular case of FatTire, an asterisk will produce the same behavior as a traditional regular expression, but translated into alternative traversing paths. FlowLog and Flog deliver different features, such as model checking, dynamic verification and stateful middleboxes. For instance, the build of a stateful firewall application is rapidly feasible. Merlin is one of the first examples of unified framework for controlling different network components, such as forwarding devices, middleboxes, and end-hosts. An important advantage is backward-compatibility with existing systems. To achieve this goal, Merlin generates specific code for each type of component. Taking a policy definition as input, Merlin's compiler determines forwarding paths, transformation placement, and bandwidth allocation. The compiled outputs are sets of component-specific low-level instructions to be installed in the devices. Merlin's policy language also allows operators to delegate the control of a sub-network to tenants, while ensuring isolation. This delegated control is expressed by means of policies that can be further refined by each tenant owner, allowing them to customize policies for their particular needs.

2.2.5.2 Network Applications

An SDN infrastructure can be deployed on any traditional network environment, from home and enterprise networks to data centers and Internet exchange points. Such variety of environments has led to the coverage of a wide array of network applications. Existing network applications perform traditional functionality such as routing, load balancing, and security policy enforcement, but also explore novel approaches, such as reducing power consumption, fail-over and reliability functionalities to the infrastructure layer, end-to-end QoS enforcement, network virtualization, mobility management in wireless networks, among many others. The variety of network applications, combined with real use case deployments, is expected to be one of the major forces on fostering a broad adoption of SDN. Despite the wide variety of use cases, most SDN applications can be grouped in one of five categories, namely in *Traffic Engineering*, *Mobility and Wireless*, *Measurement and Monitoring*, *Security and Dependency* and *Information-centric Networking*, described meticulously below.

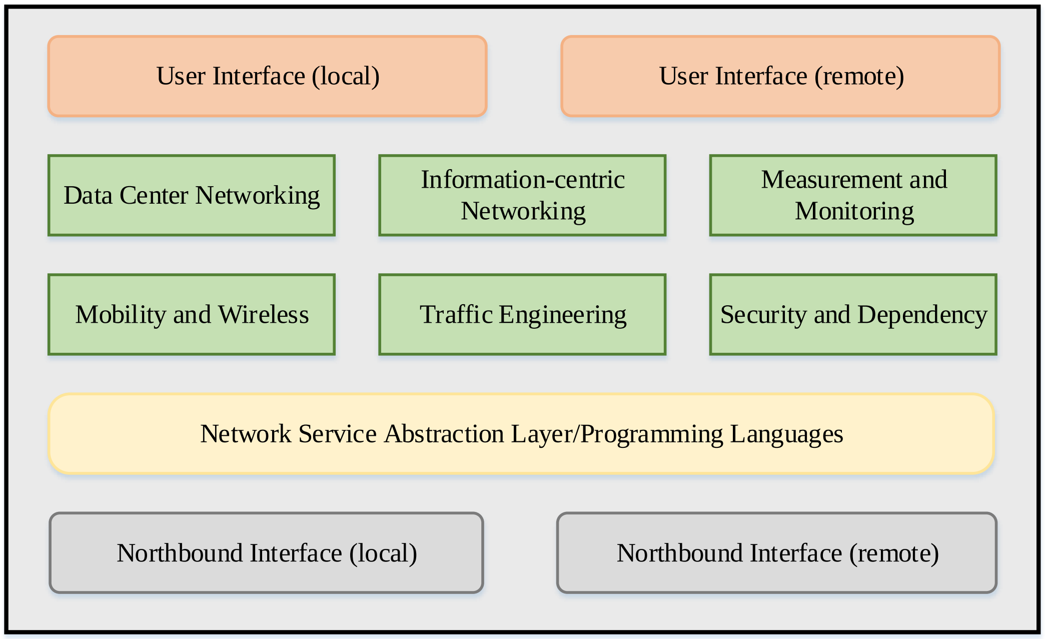

User Interface (local)
User Interface (remote)
Data Center Networking
Information-centric Networking
Measurement and Monitoring
Mobility and Wireless
Traffic Engineering
Security and Dependency
Network Service Abstraction Layer/Programming Languages
Northbound Interface (local)
Northbound Interface (remote)

Figure 2.6: *Functionalities and components of the SDN application layer*

2.2.5.2.1 Traffic Engineering Traffic engineering is a method for adaptively analyzing, regulating, and predicting the behaviour of data flowing in networks with the aim of performance optimization to meet SLAs. Traffic engineering involves establishing routing and forwarding policies based on QoS requirements. With SDN, the task of traffic engineering should be considerably simplified compared with a non-SDN network. SDN offers a uniform global view of heterogeneous equipment and powerful tools for configuring and managing network switches. Load balancing was one of the first utilized applications for SDN/OpenFlow. Different algorithms and techniques have been proposed for this purpose with the concern of scalability. A technique to allow scalability in these applications is to use *wildcard-based rules* to perform proactive load balancing [71]. Such rules can be utilized for aggregating clients requests based on the ranges of IP prefixes, allowing the distribution and directing of large groups of client requests without requiring controller intervention for every new flow.

Additionally, reactive operations are the main core as traffic bursts are detected. The controller application continuously monitors the network traffic and use threshold in the flow counters to redistribute clients among the servers when bottlenecks are likely to happen. The functionalities of SDN load-balancing also simplify the placement of network services in the network [72]. The install of a new server is followed by the load-balancing service taking the appropriate actions to seamlessly distribute the traffic among the available servers, taking into consideration both the network load and the available computing capacity of the respective servers. The monitoring is equally translated into actions in the existing southbound interfaces, used for actively monitoring the infrastructure layer load [72]. By using specialized optimization algorithms and diversified configuration options, it is possible to meet the infrastructure goals of latency, performance, and fault tolerance, while reducing power consumption, with the use of simple techniques, such as shutting down links and devices intelligently in response to traffic load dynamics. SDN also delivers a fully automated system for the management of the configuration of routers, particularly useful in scenarios that apply virtual aggregation. This implementation allows network operators to reduce the data replicated on routing tables, which is one of the causes of routing tables' growth [73]. A specialized routing application can calculate, divide and configure the routing tables of the different routing devices through a southbound API, such as OpenFlow. On top of this, traffic optimization is propelled for large scale service providers, where dynamic scale-out is required. For instance, the dynamic and scalable provisioning of VPNs in cloud infrastructures using protocols, such as ALTO, can be simplified through an SDN-based approach, since optimizing rules placement can increase network efficiency [74].

2.2.5.2.2 Mobility and Wireless Wireless networks impose a broad range of new requirements and challenges as mobile users are continuously generating demands for new services with high quality and efficient content delivery independent of location. Network providers must deal with problems related to managing the available spectrum, implementing handover mechanisms, performing efficient load balancing, responding to QoS and QoE requirements, and maintaining security. SDN can provide necessary tools for the mobile network provider and, in recent years, a number of SDN-based applications for wireless network providers have been designed. The current distributed control plane of wireless networks is suboptimal for managing the limited spectrum, allocating radio resources, implementing handover mechanisms, managing interference, and performing efficient load-balancing between cells. SDN-based approaches pose as a great opportunity for alleviating the deployment and management of different types of wireless networks, such as WLANs and cellular networks. Traditionally, hard-to-implement features are indeed becoming a reality with the SDN-based wireless networks. These include seamless mobility through efficient hand-overs, load balancing, creation of on-demand virtual access points (VAPs) [75], downlink scheduling (e.g., an OpenFlow switch can do a rate shaping or time division), dynamic spectrum usage, enhanced intercell interference coordination [76], device to device (D2D) offloading in decision of LTE transmissions per client and/or base station resource block allocations in time and frequency slots in LTE/OFDMA networks, optimization algorithms in transmission and power parameters of WLAN devices, definition and assign of transmission power values to each resource block and each base station in LTE/OFDMA networks, easy management of heterogeneous network technologies and interoperability between them, seamless subscriber mobility and cellular networks, QoS and access control policies and easy deployment of new applications. The utilization towards realizing these aforementioned characteristics in wireless networks lies in the provision of programmable and flexible stack layers for wireless networks. One of the first examples is OpenRadio [77], a software abstraction layer for decoupling the wireless protocol definition from the hardware, allowing shared MAC layers across different protocols using commodity multi-core platforms. OpenRadio can be seen as the "OpenFlow for wireless networks".

2.2.5.2.3 Security and Dependency The applications of this section include:

- **Usage of SDN functionality to improve current network security**: Although SDN presents new security challenges for network designers and managers, it also provides a platform for implementing consistent, centrally managed security policies and mechanisms for the current network infrastructure [25, 78]. SDN allows the development of SDN security controllers and SDN security applications that can provision and orchestrate security services and mechanisms

- **Address of security concerns related to the SDN usage**: Since SDN involves a three-layer architecture, new approaches are necessary in distributed control and encapsulation of data [79]. All of this introduces the potential for new vectors for attack while threats can occur at any of the three layers or in the communication between layers. SDN applications are needed to provide for the secure use of SDN itself

The SDN capabilities deliver the ability of collecting statistics data from the network and allowing applications to actively program the forwarding devices, rendering this opportunity as a paramount utility for proactive and smart security policy enforcement techniques. One example is the Active security, a novel security methodology proposing a feedback loop to improve the control of defense mechanisms of a networked infrastructure [80].

2.2.5.2.4 Information-centric Networking Information-centric Networking (ICN), also known as content-centric networking, has received significant attention in recent years, mainly directed by the fact that distributing and exploiting the circulated network information has become the major function of the Internet today. From small enterprises to large scale cloud providers, most of the existing IT systems and services are strongly dependent on highly scalable and efficient data centers. Yet, these infrastructures still pose significant challenges regarding computing, storage and networking. Concerning the latter, data centers should be designed and deployed in such a way as to offer high and flexible cross-section bandwidth and low latency, QoS based on the application requirements, high levels of resilience, intelligent resource utilization to reduce energy consumption and improve overall efficiency, agility in provisioning network resources, for example by means of network virtualization and orchestration with computing and storage, and so forth. Not surprisingly, many of these issues remain open due to the complexity and inflexibility of traditional network architectures. Unlike the traditional host-centric networking where information is obtained by contacting specified named hosts, ICN aims at providing native network primitives for an powerful information retrieval by directly naming and operating on information objects. With ICN, a distinction exists between location and identity, decoupling information for its sources. The essence of this approach is that information sources can introduce the proposition of information anywhere in the network, as the information is named, addressed, and matched independently of its location. In ICN, instead of specifying a source-destination host pair for communication, a piece of information itself is named. After the execution of a request, the network is responsible for locating the best source that can provide the desired information. Routing of information requests seeks to find the best source for the information, based on a location-independent name. Deploying ICN on traditional networks is challenging, since existing routing equipment needs to be updated or replace with ICN-enabled routing devices. Furthermore, ICN targets its delivery model from host-user to content-user, designing a need for a clarified distinction between the task of information demand and supply, and the task of forwarding. SDN has the potential of fulfilling the necessary technology for the deploy of ICN as it delivers the provision for programmability of the forwarding elements and a separation of control and data planes.

Another potential application of SDN in data centers is in detecting abnormal behaviors in network operation [81]. By using different behavioral models and gathering demanded information from the several elements involved in the operation of a data center, such as infrastructure, operators and applications, it is available to continuously build signatures for applications by passively capturing the control traffic. Consequently, the signature history can be used to identify differences in the normal behavior. When a difference is detected, operators can reactively/proactively take corrective measures, contributing in the isolation of abnormal components and avoid further damage to the infrastructure. On top of that, SDN provides the infrastructure providers with the ability of exposing networking primitives to their customers, by allowing virtual network isolation, custom addressing, and the placement of middleboxes and virtual desktop cloud applications. The exploration of the full potential of virtual networks in clouds is enabled through the essential feature of virtual network migration. Similar to the traditional virtual machine migration, a virtual network may migrate when its virtual machines move from one place to another. Integrating live migration of virtual machines and virtual networks is one of the forefront challenges.

2.2.5.2.5 Measurement and Monitoring The domain of measurement and monitoring applications can roughly be divided into two categories, the applications that provide new functionality for other networking services, and the applications that add value to OpenFlow-based SDNs. An example of the first category is in the area of broadband home connections. An SDN-based broadband home connection can simplify the addition of new functions in measurement systems, allowing the reaction to changing conditions in the home network. The second category typically involves using different kinds of sampling and estimation techniques to reduce the burden of the control plane in the collection of data plane statistics. In the literature, the are several techniques that have been applied, such as stochastic and deterministic packet sampling techniques [82], fine-grained monitoring of wildcard rules [83], traffic matrix estimation [84], two-stage Bloom filters to represent monitoring rules and many more [85, 86].

2.3 Network Function Virtualization

NFV originated from discussions among major network operators and carriers on the improvisation of the network operations in the high-volume multimedia era. These discussions resulted in the publication of the original NFV white paper [87], where the group listed as the overall objective of NFV in leveraging standard IT virtualization technology to consolidate many network equipment types onto industry standard high-volume servers, switches, and storage, which could be located in data centers, network nodes, and in the end-user premises. The white paper highlights that the source of the need for this new approach is that networks include a large and growing variety of proprietary hardware appliances, leading to the following negative consequences:

- New network services may require additional different types of hardware appliances, and finding the space and power to accommodate these boxes is becoming increasingly difficult
- New hardware means additional capital expenditures. Once new types of hardware appliances are acquired, operators are faced with the rarity of skills necessary to design, integrate, and operate increasingly complex hardware-based appliances
- Hardware-based appliances rapidly reach end of life, requiring much of the procure-design-integrate-deploy cycle to be repeated with little or no revenue benefit
- As technology and services innovation accelerates to meet the demands of an increasingly network-centric IT environment, the need for an increasing variety of hardware platforms inhibits the introduction of new revenue-earning network services

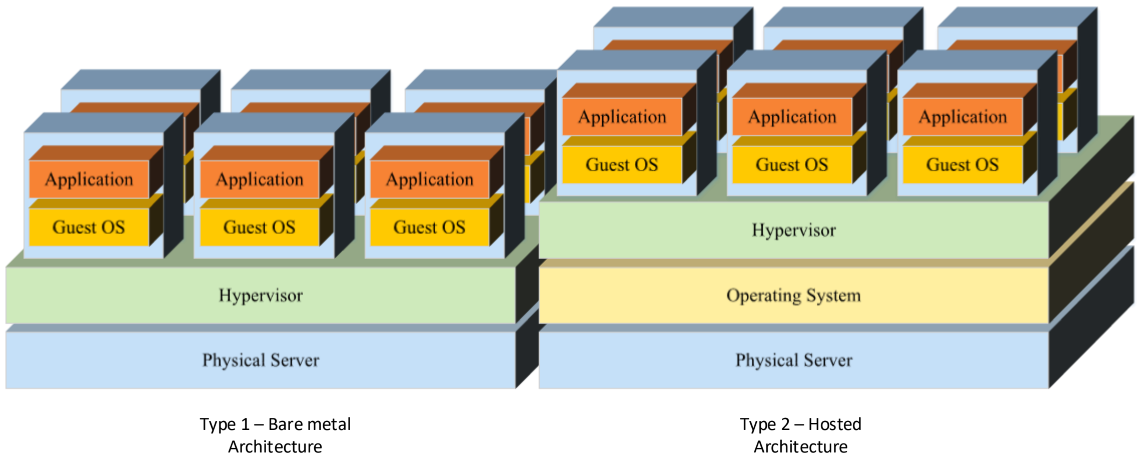

Application
Guest OS
Hypervisor
Physical Server
Operating System
Type 1 – Bare metal Architecture
Type 2 – Hosted Architecture

Figure 2.7: *Types of Virtual Machine Monitors*

The NFV approach moves away from dependence on a variety of hardware platforms to the use of a small number of standardized platform types, with virtualization techniques used to provide the needed network functionality. In the white paper, the group expresses the belief that the NFV approach is applicable to any data plane packet processing and control plane function in fixed and mobile network infrastructures.

2.3.1 Hardware and Container Virtualization

Virtualization is not a new technology. During the 1970s, IBM mainframe systems offered the first capabilities that would allow programs to use only a portion of a system's resources. Various forms of that ability have been available on platforms since that time. Virtualization came into mainstream computing in the early 2000s when the technology was commercially available on x86 servers. Organizations were suffering from a surfeit of servers because of a Microsoft Windows-driven "one application, one server" strategy. Moore's Law resulted to the rapid hardware improvements outpacing software's capabilities, and most of these servers were vastly underutilized, often consuming less than 5% of the available resources in each server. In addition, this over-abundance of servers filled data centers and consumed vast amounts of power and cooling. This issue was straining a corporation's ability to manage and maintain their infrastructure. These difficulties were relieved with the capabilities of virtualization.

The component that enables the capabilities of virtualization is a Virtual Machine Monitor (VMM), or commonly known today as a *hypervisor*. This software is stated between the hardware and the Virtual Machines (VMs) acting as a resource broker. The hypervisor allows multiple VMs to safely coexist on a single physical server host and share its resources. The number of guests that can exist on a single host is measured as a consolidation ratio. Consequently, this leads to fewer cables, fewer network switches, less floor space, cooling and number of cables. Server consolidation became a tremendously valuable way to solve a costly and wasteful problem. Today, more virtual servers are deployed in the world than physical servers, and virtual server deployment continues to accelerate. Hypervisors are distinguished in two main categories, based on whether there is another OS between the hypervisor and the physical host. A Type 1 hypervisor is loaded as a thin software layer directly into a physical host or, as commonly said, onto the "bare metal" of the physical host. Once it is installed and configured, the server can then support VMs as guests. In mature environments, where virtualiza-

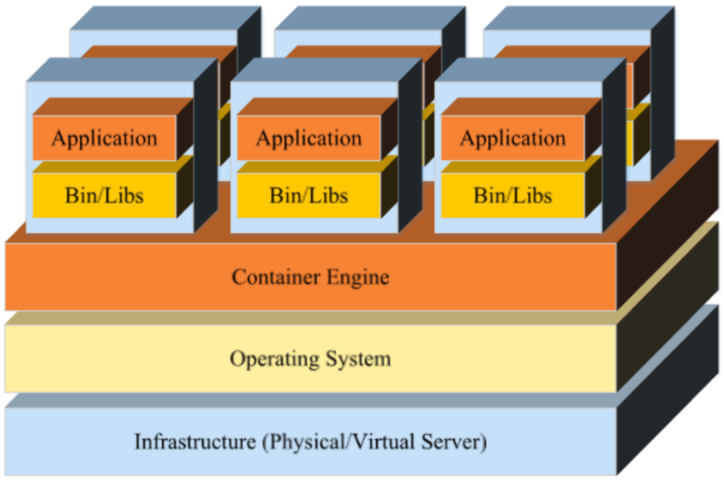


Figure 2.8: *Architecture of container virtualization*

tion hosts are clustered together for increased availability and load balancing, a hypervisor can be staged on a new host, the new host can be joined to an existing cluster, and VMs can be moved to the new host without any interruption of service. Some examples of Type 1 hypervisors are VMware ESXi, Microsoft Hyper-V, and the various open source Xen variants. However, the traditional solution that works as a traditional application with program code that is loaded on top of an OS environment also exists as this is exactly how a Type 2 hypervisor is deployed. Some examples of Type 2 hypervisors are VMware Workstation and Oracle VM Virtual Box.

Fig. 2.7 depicts the different types of VMMs. On this context, there are some important differences between the Type 1 and the Type 2 hypervisors. Typically, Type 1 hypervisors deliver improved performance than Type 2 as the former do not have the extra layer. Also, since Type 1 hypervisor doesn't compete for resources with an OS, there are more resources available on the host, and therefore, more VMs can be hosted on a virtualization server using a Type 1 hypervisor. Type 1 hypervisors are also considered to be more secure than the Type 2 hypervisors. VMs on a Type 1 hypervisor make resource requests that are handled external to that guest, and they cannot affect other VMs or the hypervisor by which they are supported. This is not necessarily true for VMs on a Type 2 hypervisor and a malicious guest could potentially affect more than itself. A Type 1 hypervisor implementation would not require the cost of a host OS, though a true cost comparison would be a more complicated discussion. Type 2 hypervisors allow a user to take advantage of virtualization without needing to dedicate a server to only that function. Developers who need to run multiple environments as part of their process, in addition to taking advantage of the personal productive workspace that a PC operating system provides, can do both with a Type 2 hypervisor installed as an application. The VMs that are created and used can be migrated or copied from one hypervisor environment to another, reducing deployment time and increasing the accuracy of what is deployed, reducing the time to market of a project.

Additionally on VMs, a relatively recent approach to virtualization is known as *container virtualization*, where software, known as a *virtualization container*, runs on top of the host OS kernel and provides an execution environment for applications. Fig. 2.8 depicts the architecture of the container virtualization. Unlike hypervisor-based VMs, containers do not aim in emulating physical servers, but instead, all containerized applications on a host share a common OS kernel, eliminating the resources needed to run a separate OS for each

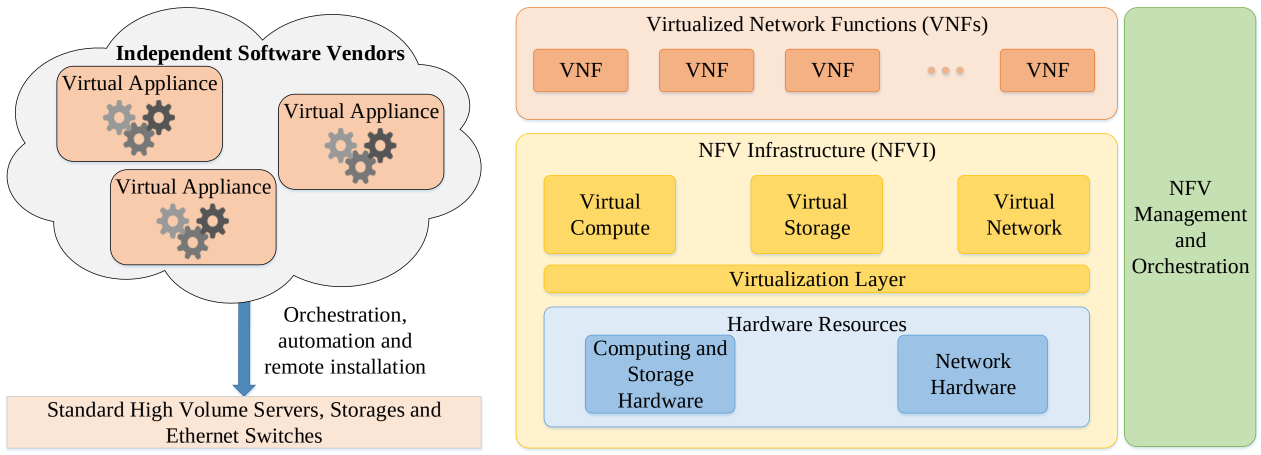

Independent Software Vendors
Virtual Appliance
Virtual Appliance
Virtual Appliance
Orchestration, automation and remote installation
Standard High Volume Servers, Storages and Ethernet Switches
Virtualized Network Functions (VNFs)
VNF
VNF
VNF
VNF
NFV Infrastructure (NFVI)
Virtual Compute
Virtual Storage
Virtual Network
Virtualization Layer
Hardware Resources
Computing and Storage Hardware
Network Hardware
NFV Management and Orchestration

Figure 2.9: *Notion of NFV from the perspective of NFV network appliance deployment (left) and high-level NFV architecture (right)*

application and can greatly reduce overhead. As the containers execute on the same kernel, containers produce smaller cost complexity compared to a hypervisor/guest OS VM arrangement.

2.3.2 Network Function Virtualization Concepts

NFV is a paramount starting point from traditional approaches to the design, deployment, and management of network services. NFV decouples network functions, such as Network Address Translation (NAT), firewalling, intrusion detection, Domain Name Service (DNS), and caching, from proprietary hardware appliances with the aim of running in software on VMs/Containers. NFV builds on standard virtualization technologies, extending their use into the networking domain. In traditional networks, all devices are deployed on proprietary/closed platforms. All network elements are enclosed boxes that cannot be shared. Each device requires additional hardware for increased capacity, but this hardware is idle when the system is running below capacity. With NFV, however, network elements are independent applications that are flexibly deployed on a unified platform comprising standard servers, storage devices, and switches. In this way, software and hardware are decoupled, and capacity for each application is increased or decreased by adding or reducing virtual resources. Created as part of the European Telecommunications Standards Institute (ETSI), the Network Functions Virtualization Industry Standards Group (ISG NFV) has the lead and indeed almost the sole role in creating NFV standards while been established in 2012 by seven major telecommunications network operators [88].

Fig.2.9 depicts a high-level view of the NFV framework defined by ISG NFV. This framework supports the implementation of network functions as software-only functionalities. Undisputedly, the core component of the NFV landscape is the virtualization of the several network functions, namely the *Virtualized Network Functions* (VNFs), comprising virtualized software implementations of traditional network functions. The deployment of VNFs takes place in the *NFV Infrastructure* (NFVI), which is the totality of all hardware and software components that build the environment where VNFs are deployed. The NFVI can span in several locations, where the network connectivity between these locations is considered as part of the NFVI. The encompassment of the orchestration and lifecycle management of physical/software resources that support the infrastructure virtualization, and the lifecycle management of VNFs is feasible through the *NFV Management*

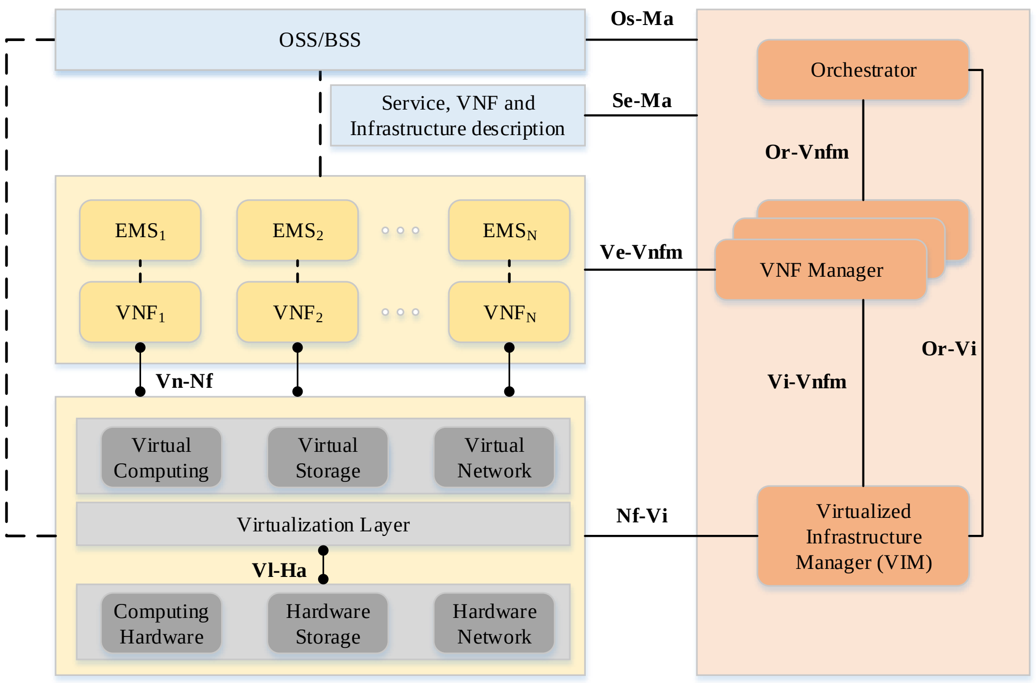


Figure 2.10: *NFV reference architectural framework*

and Orchestration (MANO). NFV MANO focuses on all virtualization-specific management tasks necessary in the NFV framework. The ISG NFV Architectural Framework document specifies that in the deployment, operation, management and orchestration of VNFs, two types of relations between VNFs are supported:

- **VNF forwarding graph (VNFFG)**: Specification of network connectivity between VNFs, such as a chain of VNFs on the path to a web server tier (for example, firewall, NAT, load balancer)
- **VNF set**: Lack of specification of connectivity between VNFs, such as a web server pool

VNFs are the building blocks used to create an end-to-end *Network Service* (NS). A VNF is modular and provides limited functionality on its own. For a given traffic flow within a given application, the service provider steers the flow through multiple VNFs to achieve the desired network functionality. This is referred to as *service chaining*. A VNF may be made up of one or more VNF components (VNFC), each of which implements a subset of the VNF's functionality. Each VNFC may be deployed in one or multiple instances, which can be deployed on separate, distributed hosts to provide scalability and redundancy. The top part of Fig.2.11 depicts a physical realization of a NS. At a top level, the NS consists of endpoints connected by a forwarding graph of network functional blocks, called *network functions* (NFs). In the Architectural Framework, NFs are viewed as distinct physical nodes. The endpoints are beyond the scope of the NFV specifications and include all customer-owned devices. So, in the figure, endpoint A could be a smartphone and endpoint B a Content Delivery Network (CDN) server. The logical links are supported by physical paths through infrastructure networks (wired or wireless).

The bottom part of Fig.2.11 shows a virtualized NS configuration that could be implemented on the physical configuration. VNF$_1$ provides network access for endpoint A, and VNF$_3$ provides network access for B. The figure also depicts the case of a nested VNFFG constructed from other VNFs, namely the VNF$_{2-A}$, VNF$_{2-B}$, and VNF$_{2-C}$. All of these VNFs run as VMs/containers on physical machines, called *points of presence* (PoPs). This configuration illustrates several important points. First, VNFFG consists of three VNFs even though ultimately all the traffic transiting between VNF$_1$ and VNF$_3$. The reason behind this is that three separate and distinct network functions are being performed. For example, it may be that some traffic flows need to be subjected to a traffic policing or shaping function, which could be performed by VNF$_{2-C}$. Thus, some flows would be routed through VNF$_{2-C}$, while others would bypass this network function. A second observation is that two of the VMs/containers in the VNFFG are hosted on the same physical machine. As these two VMs/containers perform different functions, they need to be distinct at the virtual resource level but can be supported by the same physical machine. But this is not required, and a network management function may at some point decide to migrate one of the VMs/containers to another physical machine, for performance reasons. This movement is transparent at the virtual resource level.

Fig.2.10 shows a meticulous perspective of the ISG NFV reference architectural framework and the constituent components. Firstly, the NFV MANO facility includes the following functional blocks:

- **NFV orchestrator (NFVO)**: Install and configuration of new NS and VNF packages, NS lifecycle management, global resource management, and validation and authorization of NFVI resource requests.
- **VNF Manager (VNFM)**: Monitor of lifecycle management of VNF instances.
- **Virtualized Infrastructure Manager (VIM)**: Control and management of the interaction of a VNF with computing, storage, and network resources under its authority, in addition to their virtualization.

Additionally, the EMS components represent the collection of *Element Management Systems* (EMS) that manage the VNFs. On the top layer, the *Operational and Business Support Systems* (OSS/BSS) are defined by the service provider. The NFVI together with the VIM provide and manage the virtual resource environment and its underlying physical resources. The VNF/EMS layer provides the software implementation of network functions, together with one or more VNFMs. Finally, there is a management, orchestration, and control layer consisting of OSS/BSS and the NFVO. Furthermore, several reference points are highlighted that constitute interfaces between functional blocks in the architectural framework of Fig.2.10:

- **Vn-Nf**: APIs used by VNFs to execute on the virtual infrastructure. Application developers, whether migrating existing network functions or developing new VNFs, require a consistent interface, providing functionality and the ability to specify performance, reliability, and scalability requirements
- **Nf-Vi**: Interfaces between the NFVI and the VIM, facilitating specification of the NFVI capabilities for the VIM. The VIM must be able to manage all the NFVI virtual resources, including allocation, monitoring of system utilization, and fault management
- **Vi-Vnfm**: Resource allocation requests by the VNFM and the exchange of resource configuration and state information.
- **Ve-Vnfm**: Requests for VNF lifecycle management and exchange of configuration and state information.
- **Se-Ma**: Interface between the orchestrator and a data set that provides information regarding the VNF deployment template, VNFFG, service-related information, and NFV infrastructure information models.

- **Se-Ma**: Interface responsible for the interaction between the orchestrator and the OSS/BSS.
- **Or-Vi**: Resource allocation requests by the NFVO and the exchange of resource configuration and state information.
- **Or-Vnfm**: Exchange of configuration information to the VNFM and collecting state information of the VNFs necessary for network service lifecycle management.
- **Vi-Ha**: Mapping of interfaces to the physical hardware. A well-defined interface specification will facilitate for operators sharing physical resources for different purposes, reassigning resources for different purposes, evolving software and hardware independently, and obtaining software and hardware component from different vendors.

2.3.3 NFV Management and Orchestration

The NFV MANO component of NFV has as its primary and complex function the management and orchestration of an NFV environment. Further complicating MANO functionality is its need to interoperate with and cooperate with existing OSS/BSS functionalities in providing management functionality for customers whose networking environment consists of a mixture of physical and virtual elements. As can be seen, there are five management blocks, which are three within NFV MANO, EMS associated with VNFs, and OSS/BSS. These two latter blocks are not part of MANO but do exchange information with MANO for the purpose of the overall management of a customer's networking environment. Focusing on the infrastructure, VIM comprises the functions that are used to control and manage the interaction of a VNF with computing, storage, and network resources under its authority, as well as their virtualization. A single instance of a VIM is responsible for controlling and managing the NFVI compute, storage, and network resources, usually within one operator's infrastructure domain. This domain could consist of all resources within an NFVI-PoP, resources across multiple NFVI-PoPs, or a subset of resources within an NFVI-PoP. To deal with the overall networking environment, multiple VIMs within a single MANO may be needed. A VIM performs resource management, in terms of 1) inventory of software (for example, hypervisors), computing, storage and network resources dedicated to NFV infrastructure, 2) allocation of virtualization enablers, for example, VMs onto hypervisors, compute resources, storage, and relevant network connectivity, 3) management of infrastructure resource and allocation, and operations for 1) visibility into and management of the NFV infrastructure, 2) root cause analysis of performance issues from the NFV infrastructure perspective, 3) collection of infrastructure fault information and 4) collection of information for capacity planning, monitoring and optimization

In the same context of MANO, the VNFM is responsible for the 1) instantiation, including VNF configuration if required by the VNF, 2) deployment template (for example, VNF initial configuration with IP addresses before completion of the VNF instantiation operation), 3) instantiation feasibility checking, if required, 4) instance software update/upgrade, 5) instance modification, 6) instance scaling out/in and up/down, 7) instance-related collection of NFVI performance measurement results and faults/events information, and correlation to VNF instance-related events/faults, 8) instance assisted or automated healing, 9) instance termination, 10) lifecycle management change notification, 11) management of the integrity of the VNF instance through its lifecycle, and 12) overall coordination and adaptation role for configuration and event reporting between the VIM and the EM.

The NFVO is responsible for the management and coordination of the resources under the management of different VIMs. NFVO coordinates, authorizes, releases and engages NFVI resources among different PoPs or within one PoP. This does so by engaging with the VIMs directly through their northbound APIs instead of engaging with the NFVI resources directly. Network services orchestration manages/coordinates the creation of an end-to-end service that involves VNFs from different VNFMs domains, through the creation of end-to-end service between different VNFs and coordination with the respective VNFMs so that it does not need to talk to VNFs directly. An example is creating a service between the base station VNFs of one vendor and core node VNFs of another vendor.

Network Service Descriptors (NSDs) and Virtual Network Function Descriptors (VNFDs) contain and describe all relevant aspects of NS and constituent VNFs in a structured way. The syntax and structure of these descriptors is clearly defined by an NSD-schema and VNFD-schema, respectively. The schemas ensure that all relevant information is included and can easily be parsed. Associated with NFVO, four repositories of information needed for the MANO functions:

- **Network services catalog**: List of the usable NSDs. A deployment template for a NS in terms of VNFs and description of their connectivity through virtual links is stored in NS catalog for future use.
- **VNF catalog**: Database of all usable VNF descriptors. A VNFD includes also the description of a VNF in terms of its deployment and operational behavior requirements. It is primarily used by VNFM in the process of VNF instantiation and lifecycle management of a VNF instance. The information provided in the VNFD is also used by the NFVO to manage and orchestrate network services and virtualized resources on NFVI.
- **NFV instances**: List containing details about NS instances and related VNF instances.
- **NFVI resources**: List of NFVI resources utilized for the purpose of establishing NFV services.

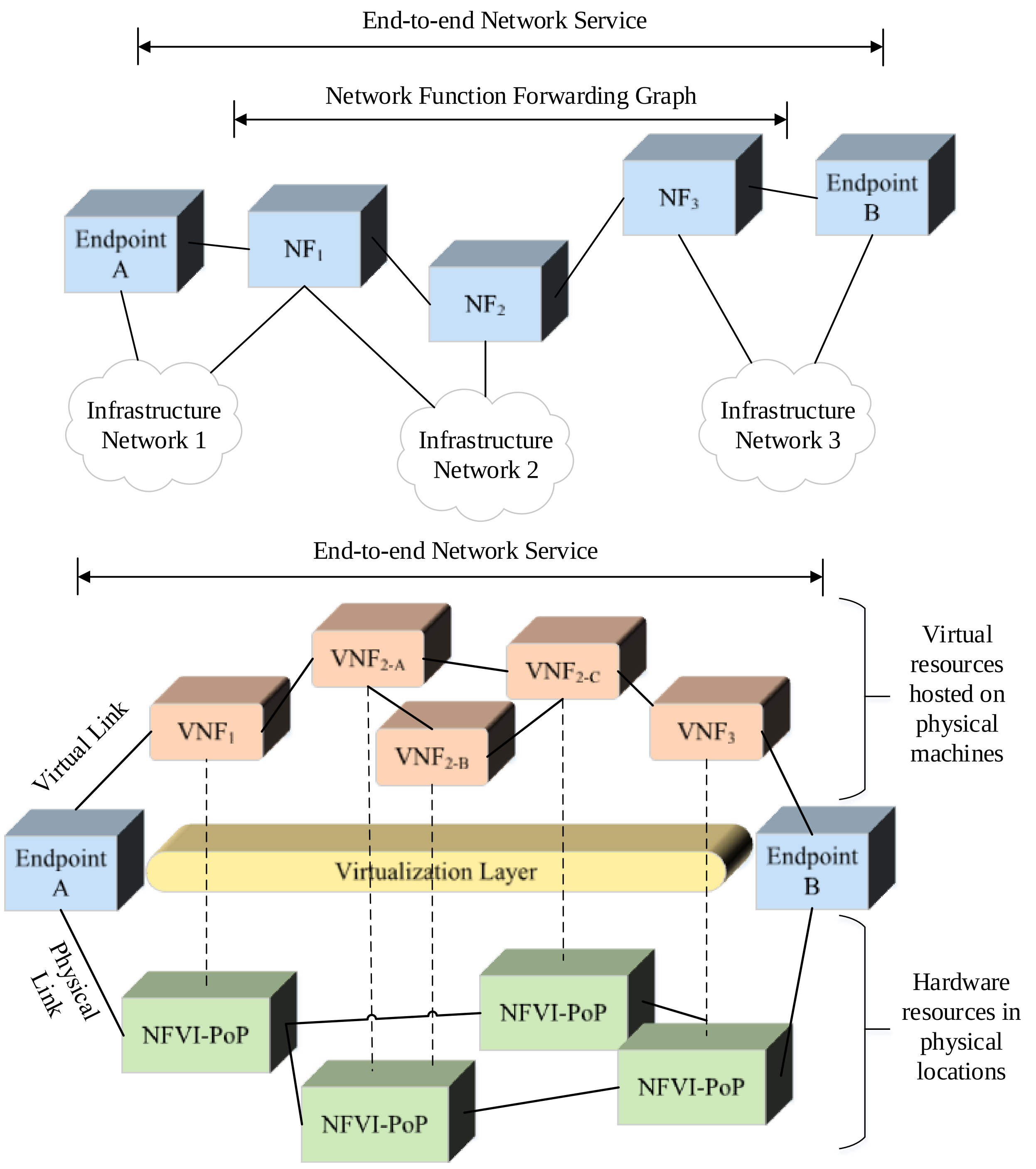


Figure 2.11: *NFV configuration of a network service. Top, graph representation of an end-to-end network service. Bottom, example of an end-to-end network service with VNFs and nested forwarding graph*

Chapter 3

Review of NoSQL Databases

The advent of the word *data* has been defined from the early years. The word *data* is driven from the Latin word *datum*, meaning *to give*. Datum or data means an unorganized fact or an event that need to be processed. Data is a representation of a fact, figure, and idea; it can be a number, word, or image. Thus, data points to qualitative or quantitative attributes of a variable or set of variables [89]. When data is processed, organized, structured or presented in a given context so as to make it useful, it is called *information*. Databases and Database Systems (DB) comprise a vital component of Information Technology in modern society. Various activities incorporate the interaction of humans with a database, such as banking and finance sector, Internet sites, social gaming, healthcare, etc. These interactions are examples included in the subset of *traditional database applications*, in which most of the stored information is either textual or numeric. Databases and database technology have had a major impact on the growing spectrum of Information and Communications Technology (ICT). It is undisputed that databases comprise a key role in almost all areas where computers are used, including business, e-commerce, social media, engineering, medicine and Affective Computing (AC). Though, the wide usage of the word *database* requires a precise definition, given below

Definition 3.2 *A database is a collection of related data, which are organized in data structures as information.*

The main usage is the management of the information, which in some cases involves from simply holding data for future retrieval to serving as the backbone for managing the lifecycle of complex financial or engineering processes. The basic software that is responsible for the management of these data is called *Database Management System* (DBMS), grouped along with large application middleware layer that accesses and regulates the stored information. DBMSs can be viewed as mediators between human beings and physical devices, depicted in Fig.3.2. Early DBMSs was based on explicit usage of File Systems (FS) and customized application software. With a gradual increase, principles, theories and mechanisms emerged that insulated database users from the details of its physical implementation. In the late 1960s, the first milestone in this direction was the introduction of *three-schema concept* [90]. This notion comprises an approach to developing information systems with three different views and conceptual modelling being considered the key to achieving data integration, as

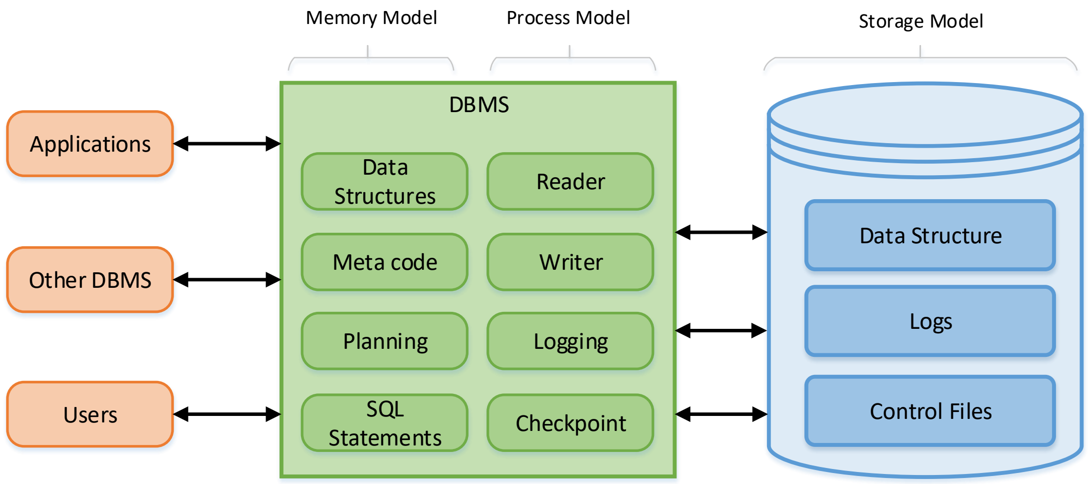


Figure 3.1: *General structure of database management system*

depicted in Fig.3.1 The key asset of this approach is the user independence of any difficult to work with the variety and complexity of data at a low level of abstraction. This architecture separated database functionalities into three different levels of abstraction

- **Internal/Physical Data Level**: The physical schema details the actual storage and the representation of data, such as files, indices, etc. on the random access disk system. It also includes the layout of files and data structures of files, such as Hash, B–tree and Flat
- **Conceptual/Logical Data Level**: The logical level represents the data schema as a set of structures from physical data level without computer-related information
- **External Data Level**: An external schema specifies a view of the data in terms of the conceptual level, accustomed to the user needs

The division of the logical and physical data level is denoted as the *Data Independence Principle* (DIP), which is undisputedly, the most important distinction between FS and DBs. The second separation of the external and logical data level is of paramount importance, since it allows different perspectives of the database that are tailored to the needs of the user. Views encrypt redundant information to the user and restructure data that is retained. However, in general, a major point is the trade-off between human convenience and reasonable performance, related with the aforementioned points. A clear example of this is the separation between logical and physical data level that the system must compile queries and regulations directed to the logical representation into actual programs. The use of the relational model became widespread only when query optimization techniques made it feasible. More generally, as the field of physical database optimization has matured, logical models have become increasingly remote from physical storage. Developments in hardware, such as gradually larger and faster memories, are also influencing the field a great deal by continually changing the limits of feasibility.

In this chapter, a general review of the NoSQL databases is provided, along with a variety of core properties from SQL databases and the evolution to the bridging to the NoSQL ecosystem. Additionally, the NoSQL

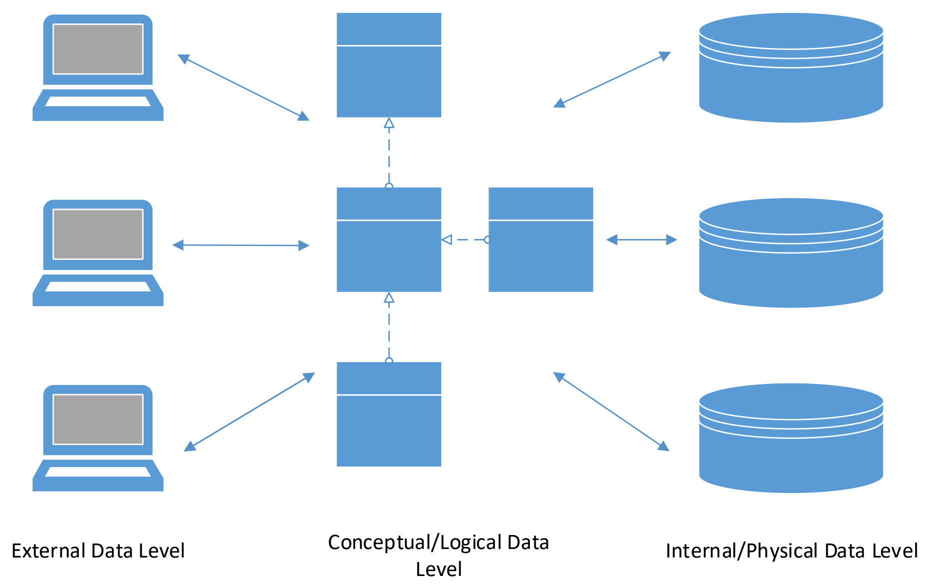


Figure 3.2: *Three schema approach*

databases, are meticulously studied, accompanied by their properties and concepts.

3.1 Concepts of the Relational Model

The relational database model was conceived by Ted Codd of IBM Research in 1970 and it attracted the immediate attention due to its simplicity and mathematical foundation [91]. This model insisted that applications should search for data by content rather than by following links. The model uses the concept of a*mathematical relation* as its basic building block, and has its theoretical basis in set theory and first-order predicate logic. The relational data model describes the world as *a collection of inter-related relations* and can be utterly defined with two terminologies, an *instance*, a table/relation with rows and columns, and a *schema*, specifying the structure including the name of the table, name and type of each column.

This study changed the way people thought about databases. In his model, the schema of the database is fully disconnected from physical information storage, rendering it as the standard principle for database systems. After learning the relational model, the first commercial implementations of the relational model became available in the early 1980s, such as the SQL/DS system on the MVS operating system by IBM and the Oracle DBMS. Since then, commercialization of relational systems laid the foundations for the DBs to become a market for business. The model has been vastly implemented in a substantial number of commercial systems, as well as a number of open source systems. Current popular commercial relational DBMSs (RDBMSs) include IBM's DB2, Oracle's Oracle, SAP's Sybase DBMS and Microsoft's SQLServer and Access. In addition, several open source systems, such as MySQL and PostgreSQL, are available. Though, SQL (Structured Query Language) was one of the first commercial languages for the Codd's relational model and it was initially developed at IBM by Donald D. Chamberlin and Raymond F. Boyce [92, 93]. This version was initially called SEQUEL (Structured English Query Language) and it was designed to manipulate and retrieve data stored in IBM's original quasi-relational database management system, *System R*, developed in IBM in 1970 [92].

The SQL language may be considered one of the major reasons for the commercial success of relational databases. Despite not entirely adhering to the relational model, it became the most widely used database language. Since SQL became a standard for relational databases, users were less concerned about migrating their database applications from other types of database systems, such as older network or hierarchical systems, to relational systems. This is based utterly on the fact that the time and cost expenses to convert to another relational DBMS product were negligible in case of users not being satisfied with the particular relational DBMS product since both systems would follow the same language standards. In practice, inconsistencies are present among various commercial relational DBMS packages. However, if the user is diligent in using only those features that are part of the standard, and if two relational DBMSs faithfully support the standard, then conversion between two systems should be simplified. Another advantage of this standard is the interoperability of the statements, where users may write statements in a database application program that can access data stored in two or more relational DBMSs without having to change the database sublanguage (SQL), as long as both/all of the relational DBMSs support standard SQL.

The relational model represents the database as a collection of relations. Informally, each relation resembles a table of values or, to some extent, a file of records. Usually, this file is called a *flat file* since each record has a simple linear or flat structure. However, there are important differences between relations and files. When a relation is thought of as a table of values, each row in the table represents a collection of related data values. A row represents a fact that typically corresponds to a real-world *entity* or *relationship*. The table name and column names are used to help to interpret the meaning of the values in each row. In the formal relational model terminology, a row is called a *tuple*, a column header is called an *attribute*, and the table is called a *relation*. The data type describing the types of values that can appear in each column is represented by a domain of possible values. The definition of the domain is given below

Definition 3.3 *A domain D is defined as a set of atomic values, where each value is indivisible.*

A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn. It is also useful to specify a name for the domain, to help in interpreting its values. The logical definition of the domain is the registration of the name, data type, and format. Additional information for interpreting the values of a domain can also be given.

3.2 Transaction Processing and ACID Theorem

Transactions are omnipresent in today's prevalent enterprise systems, providing data integrity even in highly concurrent environments. The concept of *transaction* represents the notion of a logical unit of database processing, incorporating one or several database access operations. The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language. Transaction processing systems are systems with large databases and hundreds of concurrent users executing database transactions, with the former requiring high availability and fast response time for hundreds of concurrent users.

As concurrency is deemed an essential characteristic, a challenge is born when a reference to the concurrent use of a database system is made. This statement is highly dependent on the number of users, accessing the database system. Henceforth, the two major classes of databases are 1. the *Single-user DBMS*, which are exploited only by one user, and 2. the *Multiple-user DBMS*, which can be accessed by multiple users. The latter is based on the concept of *multiprogramming*, which allows the operating system of the computer to execute multiple processes simultaneously. In this concept, two main categories of concurrent process execution exist, namely the *interleaved processing* and the *parallel processing* of concurrent transactions. As a single Central Processing Unit (CPU) can execute at most one process at a time, the former incorporates the execution of some commands from one process and suspend that process during its I/O phase to continue with some commands from the next process. The initially suspended process is resumed when it is its turn to use the CPU again. Through the latter, if the computer system has multiple hardware processors (CPUs), parallel processing of multiple processes is possible via the deployment of multiple CPUs in the multiprogramming operating systems. Fig.3.3 illustrates the two categories of interleaved and parallel transaction processing of N processes. On the left side of Fig.3.3, the processes are executed concurrently in an interleaved mode, keeping the CPU busy when a process requires an input or output (I/O) operation, such as reading a block from disk. The CPU is switched to execute another process rather than remaining idle during I/O time, thus preventing a long process from delaying other processes. On the other hand, parallel processing deploys N CPUs for the parallel execution of the processes.

Though, several challenges occur when concurrent transactions execute in an uncontrolled manner as the same program can be used to execute many different transactions on a specific database file. These challenges are listed below:

- **The Dirty Read Problem**: A failed transaction can update a specific value of an item, which can be accessed by other transactions before it is changed back to its original value. The value of the item is called *dirty data* because it has been created by a transaction that has not completed and committed yet
- **The Lost Update Problem**: The access of interleaved transactions of the same database item can render its value as incorrect
- **The Incorrect Summary Problem**: During a calculation of an aggregate function summary of a subset of the database items, transactions can update some of the items. Consequently, the aggregate functions may include updated and non updated values, rendering a stochastic result
- **The Unrepeatable Read Problem**: The sequential read value of a database item in a transaction can receive different values as it can be changed from other transactions

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for the assurance of the successful completion and the permanent record in the database. In this occasion, the transaction is said to be *committed*. In case of not being recorded, the transaction is aborted. The DBMS must not permit some operations of a transaction to be applied to the database while other operations are not as the entire transaction comprises a logical unit of database processing. If a transaction fails after executing some of its operations but before executing all of them, the already executed operations must be undone and have no lasting effect. A brief classification on the types of failures is founded below:

- **Transaction failure**: Failure of transaction due to erroneous parameter values, logical programming errors, interruption during execution and other general operations (integer overflow, division by zero, etc.)

- **Local errors or exception conditions**: Cancellation of transaction based on unknown conditions (i.e. data for the transaction may not be found)
- **System crash**: A hardware, software, or network error during the process of a transaction
- **Concurrency control enforcement**: Abort of transaction from the concurrency control method due to concurrency violations
- **Disk technical failure**: Failure in the entity of the disk due to read or write malfunction
- **Physical problems**: Failure due to an endless list of problems that are not included in the above categories (i.e. power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator)

Transactions must possess several properties, often called the *ACID properties*, that should be provided by the concurrency control and recovery methods of the DBMS. The term ACID is coined from Andreas Reuter and Theo Härder as shorthand for Atomicity, Consistency, Isolation, and Durability in 1983 [94]. This work was based on the earlier work of Jim Gray [95], who enumerated Atomicity, Consistency, and Durability, leaving out Isolation in the characterization of the transaction concept. These four properties describe the major guarantees of the transaction paradigm, which has influenced many aspects of development in database systems. The ACID properties are listed below:

1. **Atomicity**: An atomic transaction is an indivisible and irreducible series of database operations as an atomic unit of processing such that either all occur, or nothing occurs. A guarantee of atomicity prevents updates to the database occurring only partially, which can cause greater problems than rejecting the whole series outright. The assurance of atomicity is the main duty of the transaction recovery subsystem of a DBMS. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database. On the other hand, write operations of a committed transaction must be eventually written to disk.
2. **Consistency preservation**: Consistency refers to the requirement that any transaction must change affected data only in allowed ways if it is completely executed from beginning to end without interference from other transactions. In other words, it should transit the database from one consistent state to another. Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. The assurance of consistency is generally considered to be the main duty of the programmers, composing the database programs or of the DBMS module that enforces integrity constraints. Recall that a database state is a collection of all the stored data items (values) in the database at a given point in time. A consistent state of the database satisfies the obligations specified in the schema and any other existed constraints on the database . A database program should be written in order to guarantee that the database should preserve a consistent state after the complete execution of the transaction if it was in a consistent state before its execution, assuming that no interference with other parallel transactions occurs.
3. **Isolation**: Specification of how and when the implemented changes of a transaction can be visible to other parallel transactions. The initial transaction should appear as though it is being executed in isolation from

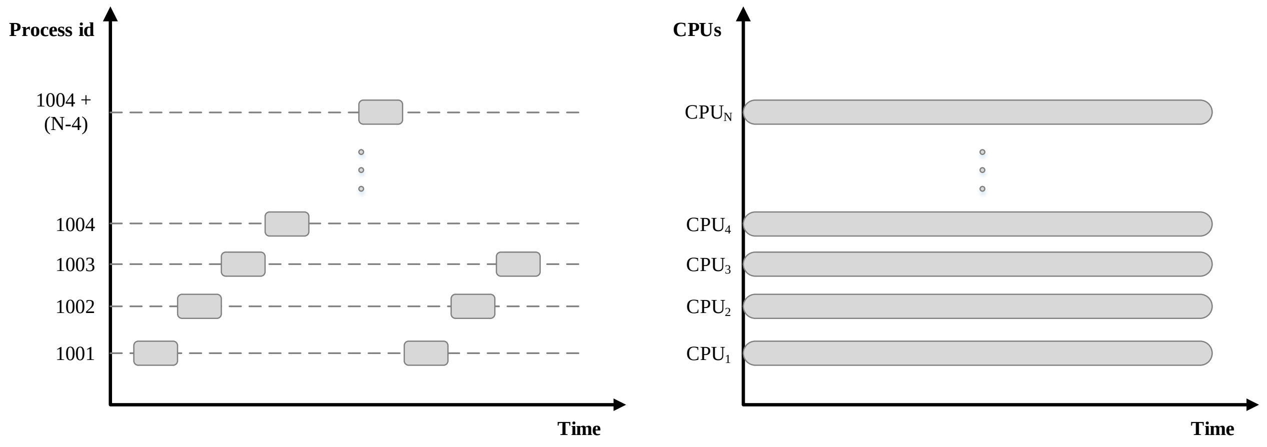


Figure 3.3: *Concurrent transaction processing in multiprogramming operating systems. Left, interleaved processing and right, parallel processing*

other transactions, even though many transactions are executing concurrently. Thus, the execution of a transaction should not be interfered. The isolation property is one of the duties of the concurrency control subsystem of the DBMS. In case of every transaction not releasing visible updates to other transactions until its commission, a level of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks. However, it does not eliminate all other problems. Thus, levels of isolations exists in the literature. A transaction is said to have level 0 isolation if it does not overwrite the dirty reads of higher-level transactions. Level 1 isolation has no lost updates, and level 2 isolation has no lost updates and no dirty reads. Finally, level 3 isolation (also called true isolation) has, in addition to level 2 properties, repeatable reads.

4. **Durability/Permanency**: The changes applied to the database by a committed transaction must persist in the database and not be lost due to any failure. The durability property is the responsibility of the recovery subsystem of the DBMS.

3.3 Theorem of CAP Trade-off

The advent of state-of-the-art technology in huge distributed systems introduces several challenges in the design of scaling mechanisms, data management techniques and storage capabilities. The reformation of such applications entails the attention to keep three core requirements in mind. Below, a key theorem in theoretical computer science is described, the *CAP Theorem*, along with the necessary theoretical background.

3.3.1 The consensus problem

In the previous years, Eric Brewer introduced the notion of a fundamental trade-off between *consistency*, *availability*, and *partition tolerance* [96]. This trade-off, which has become known as the *CAP Theorem*, has been widely discussed ever since. The CAP theorem comprises a key tool in the design of networked shared-data systems, used to detect the trade-offs in the design lifecycle. CAP theorem has influenced the design of

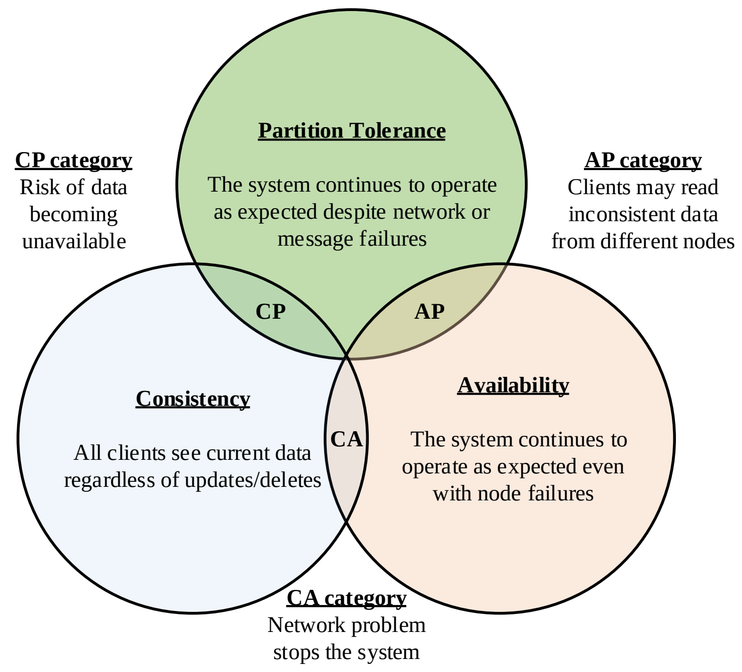

Partition Tolerance
The system continues to operate as expected despite network or message failures
CP category
Risk of data becoming unavailable
AP category
Clients may read inconsistent data from different nodes
CP
AP
Consistency
All clients see current data regardless of updates/deletes
CA
Availability
The system continues to operate as expected even with node failures
CA category
Network problem stops the system

Figure 3.4: *Visualization of CAP Theorem*

many distributed data systems, since it made designers aware of a wide range of trade-offs to consider. Though, a part of the interest in the CAP theorem derives from the fact that it conveys a more general trade-off present everywhere in the study of distributed computing and the impossibility of assuring both *safety* and *liveness* in an unreliable distributed system [97–101]. More specifically [102]:

- **Safety**: Safety refers to the subset of any unexpected procedures or events that can occur and comprise an obstacle the normal operation of a system. An algorithm is safe if nothing will stop its execution. A quiet, uneventful room is perfectly safe. The consistency characteristic of CAP Theorem is a classic safety property, as it defines that every response sent to a client is correct
- **Liveness**: On the contrary to the safety, liveness refers to the functional characteristics of a system that render it available for use. An algorithm is live if eventually something good happens. In a busy town, there may be some good things happening, and there may be some bad things happening, yet in overall, it is quite lively. The availability characteristic of CAP Theorem is a classic liveness property, where every request receives a response
- **Unreliability**: Unreliability refers to the functional characteristics of the system that define it as not able to respond to requests. There are various ways in which a system can be unreliable, such as crash failures, message loss, malicious attacks, etc

Consequently, the CAP Theorem is simply one example of the fundamental fact that both safety and liveness in an unreliable distributed system is challenging. The relationship of safety and liveness properties is under research in the field of distributed computing for many years, triggered by the first milestone of the study of Fischer, Lynch and Paterson [103]. This study focused on the fundamental problem of *consensus*, in which the achievement of overall system reliability in the presence of a number of faulty processes is studied in distributed computing and multi-agent systems. Assuming a set of processes $p_1, p_2, \cdots p_n$ and their corresponding input values $u_1, u_2, \cdots, u_n$, the problem specification of consensus is that the processes run a program to exchange their input values, when eventually the outputs of all non-faulty processes become identical, even if one or more processes fail at any time. Furthermore, the final output must belong to the set of $u_1, u_2, \cdots, u_n$, providing the equality to the value of at least one process. In [103], three requirements were proposed:

1. ***Agreement***: The final decision of every non-faulty process must be identical
2. ***Validity***: If every non-faulty process begins with the same initial value, their final decision must be equal with the input value
3. ***Termination***: Every non-faulty process must eventually decide

Therefore, agreement and validity belong to the set of the safety property, while termination to the set of the liveness property, and hence the impossibility of consensus is an important example of the inherent trade-off between safety and liveness. More specifically, the study of [103] considered a system with no partitions and potential crash failures in the processes. It concluded that consensus is impossible in such a system on the grounds that every purported consensus protocol, guaranteeing agreement and validity, there is an execution where there are no failures and the algorithm never stops. Thus, the failure of one process will affect the entire system. Immediately after the advent of this proof, a lot of interest was attracted in detecting under which conditions and assumptions it is possible to achieve both safety and liveness in systems that are sufficiently unreliable. In an attempt to answer the question with the aim of reaching consensus, researchers have focused in the perspective of *network synchrony* and what level of synchrony is necessary to avoid the inherent trade-off. A network is defined as *synchronous* under the satisfaction of the following criteria:

1. every process p_i has a clock
2. clocks are synchronized
3. every message is delivered within a fixed and known amount of time
4. every process p_i takes steps at a fixed and known rate

The case of reaching consensus in fully asynchronous systems was examined in [103] where the consensus problem is impossible to solve if even a single process crashes. By increasing the level of synchrony, *partial synchrony* is enabled when the system oscillates between asynchrony and synchrony with a dedicated time period to each state [104]. The level of synchrony was examined by Dutta and Guerraoui [105], who proved that it requires at least $k+2$ rounds for a global decision and consensus, with k denoting the maximum number of failing processes and assume that processes can fail only by crashing, even in runs that are synchronous. On the other hand, a synchronous system provides the foundations for solving consensus independent of the number of failures. In an asynchronous system, consensus is impossible for even one failure. In an partially synchronous system, consensus is feasible only if the number of failing process is less than $k/2$. If there are $k/2$ crash failures, consensus is impossible. In this case, a partitioning argument, much as in the CAP Theorem, leads to the impossibility. In real distributed management systems, designers and engineers provide techniques reaching practical consensus assuming eventually synchronous conditions [106–108].

The level of consistency has also been investigated with the aim of reaching consensus. The focus was given on the level of inconsistency through the problem of *set agreement*. More specifically, assuming p_n processes and u_n outputs, the disagreement in the output of the processes was further investigated with the aim of reaching consensus. Thus, n-set agreement defines the state in which p_n processes output all the n-different output values. It is profound that 1-set agreement is impossible with the existence of even one crash failure while n-set agreement can tolerate an arbitrary number of crash failures. It was proved that the level of consistency with n-set agreement can be solved if and only if there are at most $n - 1$ crash failures in the n processes. Consequently, n-set agreement is the higher level of agreement achieved with $n-1$ crashed processes in order to assure availability in a distributed system [97,100,101]. Furtherly developed in synchronous system, Chaudhuri *et al* analyzed the developed techniques of [100], with the degree of consistency to the running time of n-set agreement, where they proved that for t failures, at least $\lfloor t/n \rfloor + 1$ rounds are necessary and sufficient [109].

3.3.2 CAP Theorem

Brewer first presented the CAP Theorem in the context of an implemented web service by a set of servers, distributed over a set of geographically distant data centers [110]. Fig.3.4 depicts an example of the trade-off of CAP theorem on a set of distributed database management systems with services, while the terms of the CAP trade-off, namely the *consistency*, *availability*, and *partition tolerance*, are analyzed below [102]:

- **Availability**: The availability statement of the CAP Theorem defines that a system guarantees availability, through the response to a request, regardless of time constraints [111, 112]. This statement is independent from the response time of the service. From an immediate to an ultimately resulting response, CAP theorem defines an adequate response to the request. Undisputedly, a fast response is better than a slow response, yet even requiring an eventual response is sufficient to create discrepancies. In the wide spectrum of real systems, a late response is equal to a lost response, always according to the functional characteristics and requirements of the service.

- **Consistency**: The consistency statement defines that each request returns the *correct* response regardless of the server. The criterion to define *correctness* needs further examination on the grounds that service specifications differ. The differences lie in the categorization of services, from *complicated* services with complicated semantics to *trivial* and *weakly consistent* services [113–116]. The services that entail straightforward correctness requirements, through the sequential specification of the service to its semantics and atomic operations, are characterized as *simple*. A *sequential specification* defines a service in terms of its execution on a single, centralized server, which maintains some state, and each request is processed in order, updating the state and generating a response. Also, a web service is called as *atomic* if there is a single instance in between the request and the response at which the operation appears to occur for every operation. Equally, from the perspective of the client, all operations were executed by a single centralized server. In real system, there are several weaker consistency conditions, such as sequential consistency, eventual consistency, causal consistency etc. A adequate assumption for the simplest occasion of CAP theorem is the service that employs a read/write atomic shared memory, where it provides its clients with a single (emulated) register and read/write operations on that register.

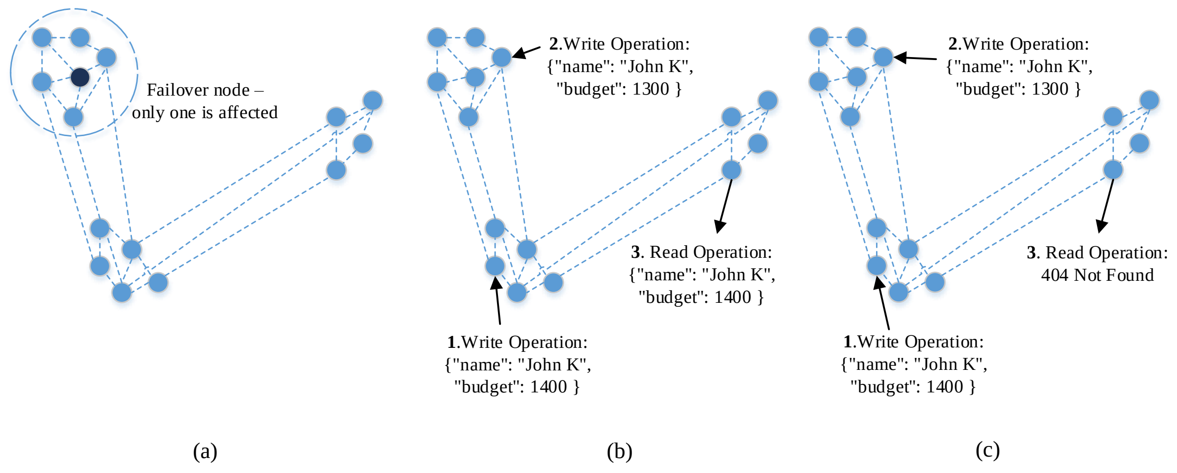


Figure 3.5: *Visualization of CAP theorem on distributed database management systems from the perspective of (a) partition tolerance, (b) consistency and (c) availability*

- **Partition tolerance**: The partition tolerance statement of the CAP theorem focuses on the internal architecture where communication among the several servers is not reliable and the servers may be partitioned into multiple groups without interaction. The criterion of partition tolerance treats communication as faulty, where messages may be delayed or, even, lost. Again, it is worth highlighting that in the wide spectrum of real systems a sufficiently delayed message is considered lost, always depending on the time requirements of the system.

The generic notion of a web service covers a plethora of applications, such as search engines, e-commerce, on-line music services and cloud-based data storage. Fig.3.5 depicts an example of the composition of the service from the $p_1, p_2, \cdots p_n$ servers with arbitrary set of clients. Fig.3.5a depicts the statement of partition tolerance, where the system continues its normal operation, sustaining any network failure that doesn't affect in failure of the entire network. This is feasible through the adequate replication of data objects across the various combinations of nodes and network with the aim of retaining the system up through intermittent outages. Additionally, Fig.3.5b depicts the statement of consistency, where all nodes see the same data at the same time and the operation starts/finishes in a consistent state. In this example, three operations are performed sequentially, two write and one read operation of the *budget* field of a specific register. In case of inconsistent state, the system will provide the first value to the read operation, being not fully informed in all nodes for the second write operation. In real systems, if a system shifts into an inconsistent state during a write operation, the entire operation is rolled back. Finally, Fig.3.5c depicts the statement of availability, where the distributed system remains continuously operational and every client receives a response, regardless of the state of any individual node in the system.

Though, the key aspects of the CAP theorem are usually misunderstood and misleading. The three properties are obviously defined in a continuous scale than binary, namely from 0% to 100%, indicating several levels of property. Even partitions have nuances, including disagreement within the system in case where a partition exists [117]. The scope of consistency declares the notion of a consistent state within a boundary, yet outside of this boundary, nothing is assured. This case is obvious in a primary partition where consistency and availability are ensured, yet the service is not available outside the partition. Well-known algorithms of such examples are Paxos

[118] and atomic multicast systems [119]. However, these algorithms are deployed to ensure global consensus, as in Chubby [120] and Megastore [121]. Thus, it is best practice to view this theorem as probabilistic guide, where all three properties are a matter of degree. For example, by choosing both consistency and availability is equal that the probability of a partition is substantially smaller than that of other systemic failures, such as disasters or multiple simultaneous faults.

In its classic interpretation, the CAP theorem ignores the factor of *latency*, which comprises a key role in the partition problem. In the normal operation, the main characteristic of CAP is denoted during a timeout, where the service is responsible for the *partition decision*, either reject the operation and decrease availability or proceed woth the operation and risk inconsistency. The re-establishment of communication introduces delays in the decision of the service. Thus, the partition property delivers a time bound, rendering the choice between consistency and availability. This notion introduces several important consequences in these terms. Firstly, there is no global notion of a partition since some nodes might detect a partition, if not all. Afterwards, the nodes detecting a partition can decide in *partition* mode and choose over consistency and availability. Consequently, the designer is responsible for setting time bounds intentionally according to target response times. Yahoo's PNUTS system introduces inconsistency by deploying asynchronous remote copies, with a master local copy server, decreasing latency. This technique delivers great results in practice since single user data is naturally partitioned according to the location of the user, with each user's data master nearby [122, 123]. Facebook deploys different technique where the master copy server is in one location. The remote user has a closer but potentially stale copy, where the update of a page goes to the master copy directly as do all the user's reads for a short time, despite higher latency. After 20 seconds, the user's traffic reverts to the closer copy, which by that time should reflect the update [117, 124].

3.4 NoSQL Ecosystem

The business ecosystem is undergoing massive change as industry moves to the Digital Economy, powered by the Internet and several other 21st century technologies, namely Cloud Computing, Mobile Computing, Social Media and Big Data. At the heart of this Digital Economy business, Web and IoT applications comprise the primary way of companies interacting with customers today, and how companies run more and more of their business. The experiences that companies deliver largely determine the user QoS/QoE via those applications. A subset of the main requirements of such applications are defined below

- Support of large numbers of concurrent transactions.
- Availability and delivery of highly responsive experiences to a global distribution of transactions.
- Handling of semi-structured and unstructured data.
- Rapid adaptation of different business requirements through frequent updates and new features.

The development of web, mobile, and IoT applications has created a new set of technology requirements, summarized in Table 3.1. The new enterprise technology architecture requires far more agile characteristics than ever before, and requires an approach to real-time data management that can accommodate unprecedented levels of scale, speed, and data variability. Many companies and organizations are faced with applications that

Table 3.1: *Set of business and technical requirements addressed from the NoSQL ecosystem*

Business Requirements	**Technical Requirements**
Seamless Internet Connectivity	Support of different entities through various data structures Update of homogeneous hardware/software architectures Support continuous streams of real-time data
Support of evergrowing Big Data	Storage of tremendous amounts of semi-structured/unstructured data Simultaneous storage of different types of data from homogeneous sources
Cloud-oriented application	On-demand scaling to support more data Software operation on a global scale with worldwide access Minimization of infrastructure costs with faster time to market
Mobility-based infrastructure	Synchronization of mobile data with remote databases in the cloud Support of various mobile platforms with a single backend
Consistent and high performance User Experience	Scaling to support millions of users Maintainability of availability 24 hours a day, 7 days a week

store vast amounts of data, introducing challenges in their management. Through a structured relational SQL system, this aim cannot be accomplished based on two main reasons, 1. SQL systems offer too many services (powerful query language, concurrency control, etc.), which adds overhead in seamless time requirements and 2. a structured data model, such the traditional relational model, which is too restrictive. Although newer relational systems do have more complex object-relational modelling options, they still require development in the schema structure.

These challenges rendered relational databases deficient in synchronization with the flow of information demanded by the users, accompanied by the larger variety of data types that occurred from this evolution. Therefore, enterprises turned to the development of *non-relational databases*, often referred to as *NoSQL*. In 1998, Carlo Strozzi [125] used the term NoSQL to name his lightweight, open source relational database that did not expose the standard SQL interface. Eric Evans, a Rackspace employee, reintroduced the term NoSQL when Johan Oskarsson of Last.fm wanted to organize an event to discuss open source distributed databases . The name attempted to label the emergence of a growing number of non-relational, distributed data stores that often did not attempt to provide ACID guarantees, which are the key attributes of classic relational database systems. The characteristics of NoSQL systems are the key points that denote the differentiation from traditional SQL systems, which are divided into two main categories described below [126]:

1. **Characteristics related to distributed databases and distributed systems**. The interconnection of multiple databases, which are spread physically across various locations and communicate via a computer network, provide a plethora of advantageous characteristics:

   • ***Scalability***. Mainly in NoSQL systems, horizontal scalability is generally used, where the distributed system is expanded by the accumulation of nodes for data storage and processing with the gradual increase of the data volume [127]. On the other hand, vertical scalability refers to the expansion of the computing and storage resources of the existing nodes. The horizontal scalability is feasible aligned with the operational mode of the database. Thus, methods and techniques are necessary for distributing the existing data among new nodes without interrupting the system operation.

- ***Availability, Replication and Eventual Consistency***: The diverse applications, incorporating NoSQL systems, require continuous system availability. Data availability is enabled through replication over two or more nodes in a transparent manner. Thus, in case of one node failing, the data is still available on other nodes. Also, one advantage of data replication is the improvement of the read performance, on the grounds that read requests can often be serviced from any of the replicated data nodes. However, write performance becomes more cumbersome, since write requests must be applied to every copy of the replicated data items. Consequently, this execution adds time overhead in write operation performance if serializable consistency is required.

- ***Replication Models***: Data replication is the concept of having geo-distributed replicated data, within a system, preferably through a non-interactive, reliable process. In traditional RDBMS databases, implementing any sort of replication is a unpropitious procedure since these systems were not developed with horizontal scaling in mind. Instead, these systems can be backed up via a semi-manual process, where live recovery would not comprise an issue. The two major replication models, which are used in NoSQL systems, namely the *master-slave* and *master-master* replication, are depicted in Figure 3.13. The former replication model requires a copy to be the master copy, receiving all write operations, and, afterwards, any change of this copy must be propagated to the slave copies, usually using eventual consistency (the slave copies will eventually be the same as the master copy). For reading operations, the master-slave model can be configured in various ways. One configuration requires all reads to also be at the master copy, so this would be similar to the primary site or primary copy methods of distributed concurrency control with similar advantages and disadvantages. Another configuration of this model allows reading operations at the slave copies. Though, this configuration does not assure that the latest written value is retrieved, on the grounds that writing operation can be done to the slave nodes after they are applied to the master copy. On the other hand, the master-master replication allows reading and writing operations at any of the replicas. However, this model does not assure that reading operations at nodes see the same values since they store different copies. Also, different users have the ability to write the same data item concurrently at different nodes of the system with the immediate issue where the values of the item will be temporarily inconsistent. A reconciliation method to resolve conflicting write operations of the same data item at different nodes must be implemented as part of the master-master replication scheme.

- ***High-Performance Data Access***: In the diverse applications rendering NoSQL database as a necessity, it is vital to retrieve individual records or objects from among the millions of data records or objects in a file. In order to achieve this, two methods exist, namely the *hashing* and *range partitioning on object keys*. The majority of accesses to an object will be equal to the provision of the key value rather than by using complex query conditions. In hashing method, a hash function $h(k)$ is applied to the key k, which denotes the location of the file. In range partitioning, a location keeps dedicated objects for a range of key values. In applications where range queries is required and multiple objects within a range of key values are retrieved, range partitioned is preferred. Otherwise,

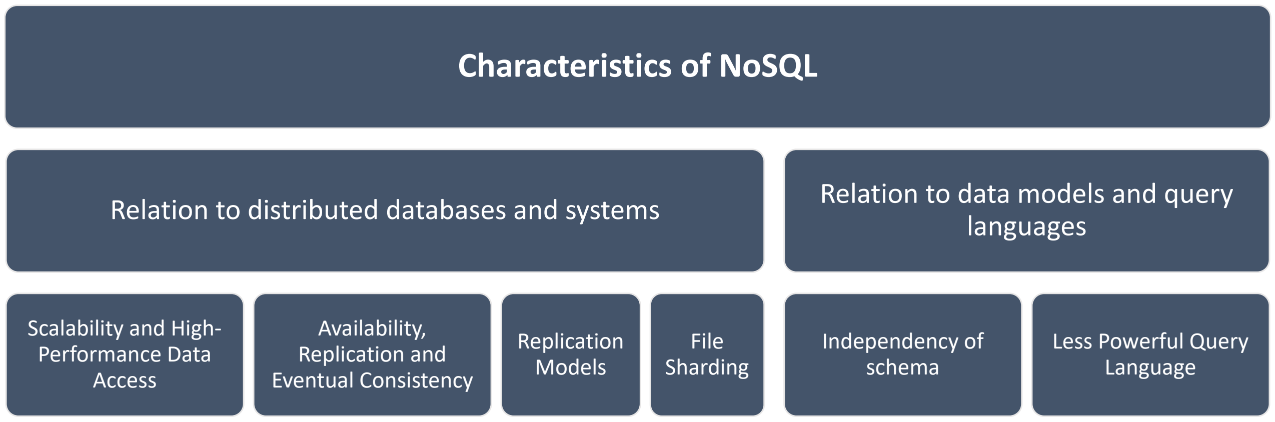

Characteristics of NoSQL
Relation to distributed databases and systems
Relation to data models and query languages
Scalability and High-Performance Data Access
Availability, Replication and Eventual Consistency
Replication Models
File Sharding
Independency of schema
Less Powerful Query Language

Figure 3.6: *Characteristics of NoSQL Databases*

hashing model is preferred when attribute conditions are different.

2. **Characteristics related to data models and query languages**: Data models are the most important part of developing software programs since they have such a profound effect not only on how the software is written, but also on how the users face the diverse tasks. NoSQL systems emphasize performance and flexibility over modeling power and complex querying with two main parallel verticals:

   - ***Independence of schema***: The flexibility of independence of the schema is an integral part of many NoSQL systems by allowing the storage of semi-structured, self-describing data. Despite the fact that the users can specify a partial schema to improve storage efficiency, generally the definition of a schema is redundant in the NoSQL systems. As there may not be a schema to specify constraints, any constraints on the data would have to be programmed in the application management programs. There are various languages for describing semi-structured data, such as JavaScript Object Notation (JSON) and Extensible Markup Language (XML). JSON is used in several NoSQL systems, but other methods for describing semi-structured data can also be used.
   - ***Less Powerful Query Languages***: The applications that incorporate NoSQL systems may not require a powerful query language on the grounds that the retrieval queries often locate single objects in a single file based on their object keys. In general, the NoSQL systems provide a set of functions and operations in the management system through APIs. Thus, retrieval and write operations of data objects is accomplished by calling the appropriate operations by the developer. Among the available NoSQL systems, some applicants provide a high-level query language, yet it delivers a subset of querying capabilities of SQL. In particular, NoSQL systems enable join operations as part of the application program and not as part of the query language itself.

3.4.1 BASE Principles

NoSQL databases embrace situations where the ACID model is overkill or would, in fact, hinder the operation of the database. NoSQL relies upon a softer model known, appropriately, as the *BASE model*. This model accommodates the flexibility offered by NoSQL and similar approaches to the management and curation of unstructured data. BASE is diametrically opposed to ACID. Where ACID is pessimistic and forces consistency at the end of every operation, BASE is optimistic and accepts that the database consistency will be in a state of flux. Although this sounds impossible to cope with, in reality it is quite manageable and leads to levels of scalability that cannot be obtained with ACID [128]. BASE consists of three principles:

1. **Basically Available**: Guaranty of data availability even in the presence of multiple failures, in terms of the CAP theorem. It achieves this by using a highly distributed approach to database management. Instead of maintaining a single large data store and focusing on the fault tolerance of that store, NoSQL databases spread data across many storage systems with a high degree of replication. An unexpected event where a failure disrupts the access to a subset of data, does not necessarily lead in a complete database outage.

2. **Soft State**: BASE databases abandon the consistency requirements of the ACID model pretty much completely. The copies of a data item may be inconsistent and the state of the system may change over time, even without input. One of the basic concepts behind BASE is that data consistency is the developer's problem and should not be handled by the database.

3. **Eventual Consistency**: Regarding consistency, the only requirement in NoSQL databases is the data convergence to a consistent state, at some point in the future, without any predefined time guarantee of this occurrence. This statement is completely differentiated from the immediate consistency requirement of ACID that prohibits a transaction from executing until the prior transaction has completed and the database has converged to a consistent state. The updates propagate as soon as there are no updates over a certain period of time defied for a particular application. Hence, the systems where BASE is the key requirement for reliability, the potential of the data changes essentially slows down.

In order to depict the advantages and the relaxation provided by the BASE requirements, the key differences of BASE and ACID must be explained. In the ACID theorem, the ultimate point is consistency with database vendors recognizing the need for partitioning databases. This was followed by the consistency assurance of the two-phase commit (2PC) technique for providing ACID guarantees across multiple database instances. The 2PC technique is a specific type of Atomic Commitment Protocol (ACP) and is a distributed algorithm that coordinates all the processes that participate in a distributed atomic transaction on whether to commit or abort (roll back) the transaction. The protocol comprises a specialized type of consensus protocol and achieves its goal even in many cases of temporary system failure (involving either process, network node, communication, etc. failures). The protocol consists of two phases:

1. **Commit-request Phase/Voting Phase**: a coordinator process attempts to prepare all the transaction's participating processes (named participants, cohorts, or workers) to take the necessary steps for either committing or aborting the transaction and to vote, either ”Yes”: commit (if the transaction participant's local portion execution has ended properly), or ”No”: abort (if a problem has been detected with the local portion).

2. **Commit Phase**: Based on voting of the participants, the coordinator decides whether to commit (only if all have voted ”Yes”) or abort the transaction (otherwise), and notifies the result to all the participants. The

participants then follow with the needed actions (commit or abort) with their local transactional resources (also called recoverable resources; e.g., database data) and their respective portions in the transaction's other output (if applicable).

Scaling systems to dramatic transaction rates requires a new way of thinking about managing resources. The traditional transactional models are problematic when loads need to be spread across a large number of components. Decoupling the operations of ACID requirements and performing them in turn provides for improved availability and scale at the cost of consistency. BASE provides a model for thinking about this decouplement. In this context, BASE properties allow for availability in a partitioned database, while challenges offering consistency have to be identified. This is often obscure on the grounds that the tendency of both business stakeholders and developers is to assert that consistency is paramount to the success of the application. Temporal inconsistency is difficult not to be obvious from the user and, thus, both engineering owners must be involved in picking the opportunities for relaxing consistency. Such techniques are provided for the relaxation of consistency through the slight delay of the provided results, for example, the caching of results. These examples are prevalent and not uncommon, while in fact people encounter this delay between a transaction and their running balance regularly (e.g., ATM withdrawals and cellphone calls). Consistency between the updates is not guaranteed while a failure between some transactions will result in the information being permanently inconsistent. One solution of ensuring a failover is a message queue through several choices for implementing persistent messages [129]. However, the design of the queue is deemed crucial, as it needs the assurance that the backing persistence is on the same resource as the database and independent. By queuing a persistent message within the same transaction as the insert, the information needed to update the running balances on the user has been captured. The transaction is contained on a single database instance and therefore will not impact system availability [128].

3.4.2 Categorization of NoSQL Databases

NoSQL databases provide the quality performance, scalability, and flexibility required of modern applications. Yet, that is where the similarity between NoSQL systems end. The only thing most NoSQL databases have in common is that they do not follow a relational data model. In further detail, NoSQL databases can be classified using different approaches and various criteria. Researchers classify them according to their data model and most of them outline four major types of NoSQL databases, namely the *key-value*, the *document-oriented*, *column-oriented* and the *graph* model databases [130]. Below, a meticulous study is provided for each category.

3.4.2.1 Key-value Databases

A hash table or an associative array is the simplest data structure that can hold a set of key/value pairs. Such data structures are extremely popular because they provide a very efficient, big O(1) average algorithm running time for accessing data [131]. The key of a key/value pair is a unique value in the set and can be easily looked up to access the data. From a data model perspective, key-value stores are the most basic type of non-tabular database. Key-value databases is a data storage model designed for storage, retrieval and managing associative arrays and are similar to maps or dictionaries where data is addressed by a unique key, as depicted in Fig.3.7.

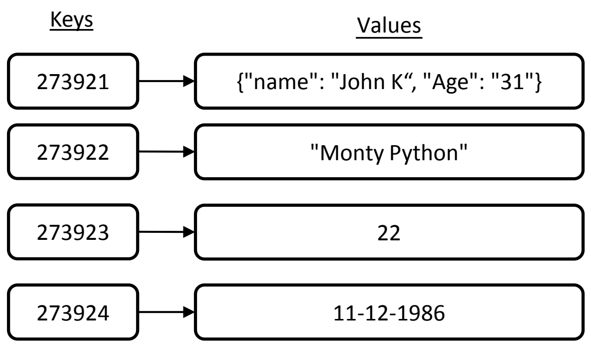


Figure 3.7: *Data structure in key-value databases*

This type of databases can include various types of data as values, which will be translated into byte arrays. Since values are translated to uninterpreted byte arrays, they are completely opaque to the system and keys are the only way to retrieve stored data. Since values are isolated and independent from each other, relationships must be handled in application logic [132]. Due to this issue, key-value databases are completely schema-free, useful for representing polymorphic and unstructured data. The addition of new values of any kind is feasible at runtime without conflicting any other stored data and influencing system availability. The grouping of key-value pairs into the collection is the only way of providing any kind of structure to the data model. The main advantages of a key-value model include:

- **Flexible data modeling**: The opacity of the values offers tremendous flexibility for modeling data to match the requirements of the application
- **Operational simplicity**: Key-value databases are designed to simplify operations by ensuring the convenience of adding and removing capacity
- **High performance**: Key-value architecture can be more efficient in many scenarios on the grounds that there is no ability of performing lock, join, union, or other operations when working with objects. The retrieval of objects with the key delivers simplicity and immediate response
- **Massive scalability**: Most key-value databases scale out efficiently on demand, using commodity hardware, through virtual operation without significant redesign of the database

The most protuberant key-value databases are Redis [133], AWS DynamoDB [134] and Project Voldemort [135]. The key-value databases are useful for conventional operations with the dependency of key attributes only. For example, in cases where acceleration of speed in web applications is essential, calculations can be performed and served quickly through key retrieval. Most key-value databases hold their dataset in memory and they are used for caching of more time-intensive queries, reducing disk I/O operations and boosting performance [136]. Caching is a popular strategy employed at all levels of a computer software stack to boost performance, from operating systems, databases, middleware components to applications. Cache systems could be rudimentary map structures or robust systems with a cache expiration policy. Thus, key-value databases pose as a great candidate for this task. Key-values model databases are generally useful for storing session information, user profiles, preferences, shopping cart data, etc.

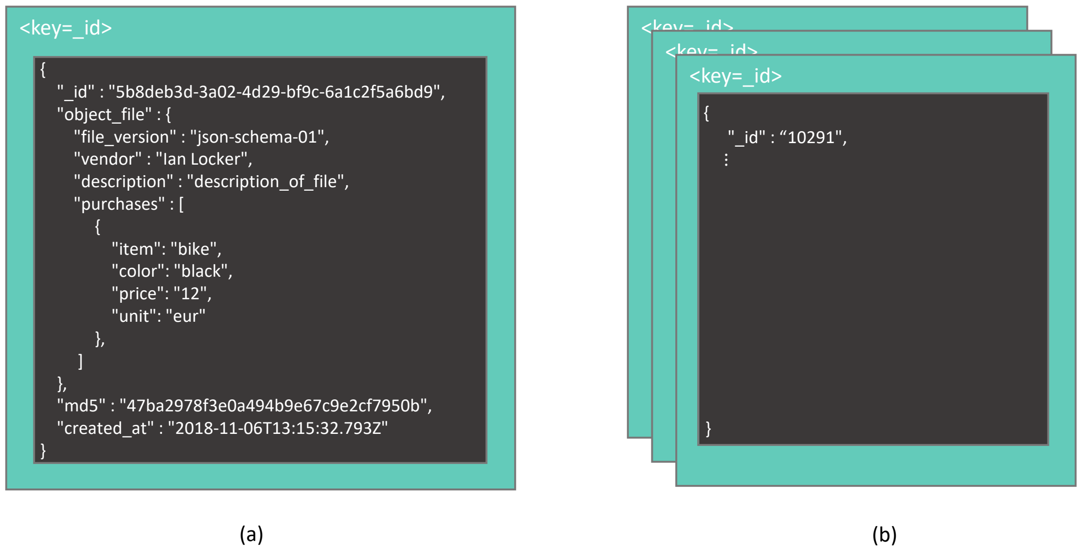
Table 3.2: *Terminology mapping of relational databases with the document-oriented databases*

RDBMS	**Document-Oriented Databases**
Database instance	Document-oriented instance
Schema	Database
Table	Collection
Row	ID
Column	Fields

3.4.2.2 Document-oriented Model Databases

Document-oriented databases are recognized as a powerful, flexible and agile storage tool of Big Data. Document-oriented databases were introduced to satisfy the storage and the management necessities of large-scale documents, encoded in a standard data exchange format, such as XML, JSON and Binary JSON (BSON). As compared to relational databases, data are first categorized into a number of predefined types, and tables are created to hold individual entries of each type. The tables define the data within each record's fields, implying that each record has the same overall form. The designer defines the relationships between the tables, selects certain fields that will be most commonly used for searching and defines indexes on them. On the contrary, a document-oriented database does not contain an internal structure that maps directly onto the concept of a table, and the fields and relationships generally don't exist as predefined concepts. Instead, all of the data for an object is placed in a single document and, consequently, it is stored in the database as a single entry. Table 3.2 depicts the terminology mapping of relational and document-oriented databases. Documents are analogous to records or rows of the relational database table. These documents are self-describing, hierarchical-tree data structures which can consist of maps, collections, and scalar values, as depicted in Figure 3.8a. Instead of tables in relational databases, a document-oriented store introduces the notion of *collections*, which are groups of documents. Depending on implementation, a document may be enforced to live inside one collection, or may be allowed to live in multiple collections.

Document-oriented databases handle document storage via the encapsulation of key-values, avoiding the split of a document into its constituent name/value pairs. Every document contains a special unique key within a collection of documents, identifying each document explicitly, along with the availability of including multiple key-value pairs, key-array pairs, and embedded documents. Figure 3.8 depicts an example of a document in JSON format and the corresponding document store structure in the database-oriented store. Though, document-oriented databases are different from key-value stores. In fact, while the search mechanism of the key-value stores functions through the retrieval of data only by key-value keeping values opaque to the system, document-oriented databases allow users to search for data based on the content of documents. Thus, users can query the content of the documents by keys, values or examples and/or their metadata objects more conviniently, delivering great flexibility required in various use cases [137]. The storage of data in interpretable JSON documents



```
<key=_id>
{
  "_id" : "5b8deb3d-3a02-4d29-bf9c-6a1c2f5a6bd9",
  "object_file" : {
    "file_version" : "json-schema-01",
    "vendor" : "Ian Locker",
    "description" : "description_of_file",
    "purchases" : [
      {
        "item": "bike",
        "color": "black",
        "price": "12",
        "unit": "eur"
      },
    ]
  },
  "md5" : "47ba2978f3e0a494b9e67c9e2cf7950b",
  "created_at" : "2018-11-06T13:15:32.793Z"
}
```

(a)

```
<key=_id>
<key=_id>
<key=_id>
{
  "_id" : "10291",
  ⋮
}
```

(b)

Figure 3.8: *Documents in a document-oriented database considering (a) an individual JSON sample with the key and its value and (b) the organization of the documents in a collection of the store*

have the further advantage of supporting data types, making document databases developer-friendly.

Document-oriented are general purpose, useful for a wide variety of applications due to the flexibility of the data model, the freedom of querying on every field and the natural mapping of the document data model to objects in modern programming languages. The storage in document-oriented stores offer multi-attribute lookups on records, having diverse kinds of key-value pairs. These systems pose as a convenient solution in applications of data integration and schema migration tasks, such as real-time analytics, storage layer of small and flexible web entities and logging. The most protuberant open-source document-oriented databases are MongoDB [138], CouchDB [139] and Couchbase [140]. All of the aforementioned databases have many common features, such as replication for fault-tolerance, volatile memory file system for data storage and MapReduce paradigm for data processing. However, CouchDB is not adapted to extremely changing data, since it requires to set up predefined queries. On the other hand, MongoDB is suitable for dynamic queries and ensures a better performance on loaded databases.

3.4.2.3 Column Family Model Databases

Column Family Databases are known as column-oriented stored, extensible record and wide columnar databases [141]. The main insipiration of the advent of the columnar databases is the Google's Bigtable, characterized as a *distributed storage system for managing structured data that is designed to scale to a very large size* [142]. In particular, Google tasks are characterized with high throughput and latency-sensitive data serving. The corresponding data model is described as *is a sparse, distributed, persistent multi-dimensional sorted map* [142]. These were created to store and process very large amounts of data distributed over various machines. Columnar databases are very similar on the surface to relational databases, yet they are actually quite different beast. The main difference is the storage of data in columns in columnar databases than in rows in the relational ones [143]. Columnar databases use the concept of *keyspace*, which contains all the column families. Figure 3.9 depicts the internal structure of the keyspace. A column family consists of multiple rows, each of

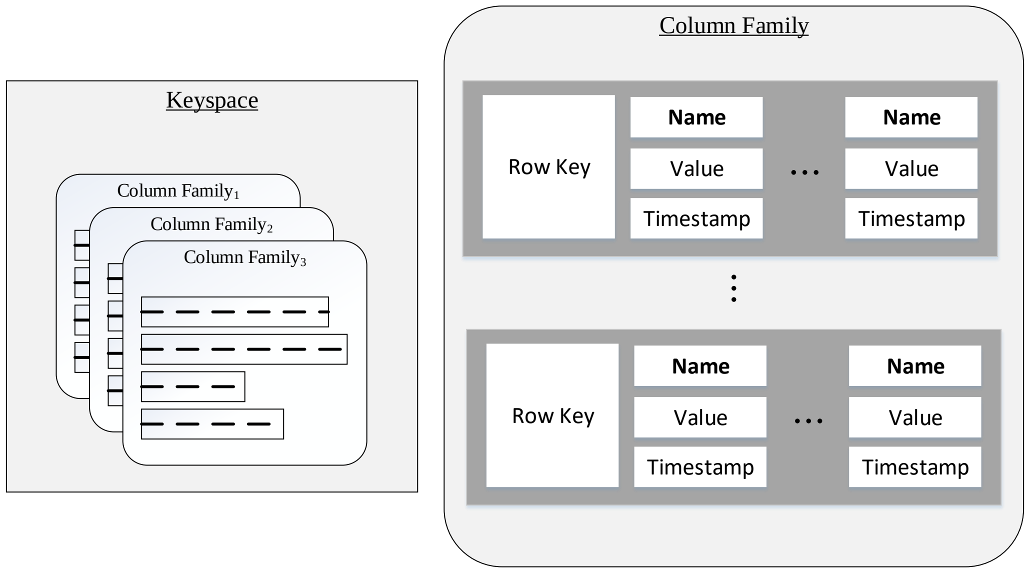


Figure 3.9: *Internal data structure in the columnar databases. On the left, the keyspace contains column families and on the right, a column family containing a variable number of rows, each one containing its own variable set of columns.*

which includes a different number of columns to the other rows. The number of columns per row is independent from the number of rows assigned to other columns. Similarly, columns are independent from each other as they can have various column names, data types, etc. Each column is contained to its row only, differentiating from the potential expansion in all rows in a relational database [144]. Each column contains a name/value pair, along with a timestamp. Each row includes a unique identifier as *row key* and each column contains a *name*, *value* and a *timestamp*, which represent the name/value pair and the date and time insertion of the data respectively. The latter is used with the aim to determine the most recent version of data. Figure 3.10 depicts the internal structure of the data stored of a column in the columnar databases. The column structure provides a variety of advantages in the performance of the columnar databases [145]:

- **Immediate Aggregate Query Execution**: The execution of queries is extremely fast since the same type of data are gathered in a specific column. More specifically, in the example of Figure 3.10, an aggregation query on the *budget* field can only concentrate on the values of this predetermined column by skipping over all the non-relevant data very quickly

- **Compression feasibility**: Since each column has one specific datatype and each row can have different columns, the data compression is delivering substantially enhanced performance than the relational databases

- **Scalability**. Columnar databases are well suited to massively parallel processing, which involves having data spread across a large cluster of machines

- **Fast loading**. Columnar stores can be loaded extremely fast with huge row tables in few seconds

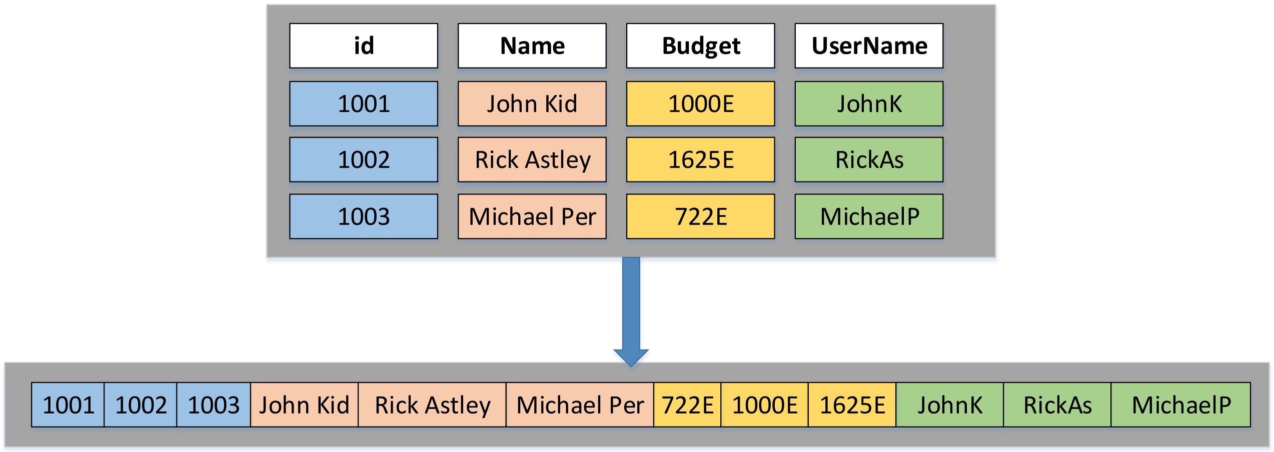


Figure 3.10: *Internal data storage of columns in columnar databases*

3.4.2.4 Graph Model Databases

Graph Model Databases use graph structures with nodes, edges and properties to represent data [146]. In essence, the model of data is represented as a network of relationships between specific elements. In contrast to the already-introduced key-oriented NoSQL databases and the relational databases ecosystem, graph database are specialized on the efficient management of heavily linked data. Although the graph model can be counter-intuitive and time-consuming, it is useful for a specific class of queries. More specifically, the applications of graph databases are incorporated in cases where traversing relationships is core element, such as social network connection, spatial data, network topologies, recommendation engines and ontologies or supply chains. Thus, the above applications are more suited for graph databases, since cost intensive operations, like recursive joins, are replaced with efficient traversals.

There are few different approaches in representing data with components in a graph database. Among the available approaches, one candidate is the *property graph*. In graph theory, the property graph is a property of graphs that depends only on the abstract structure and is independent from the graph representations, such as particular labels or drawings of the graph [147]. The basic components of the property graph model are the following:

- **Nodes**: Entities in the graph, which can have an arbitrary number of key-value pairs, namely their *properties*. Nodes can be attached with labels, representing their different roles in the environment, denoting discrete objects such as a person, a place, or an event. Also, node labels serve to attach metadata (such as index or constraint information) to certain nodes

- **Relationships**: Directed, named, semantically-relevant connections between two nodes. A relationship always has a direction, a type, a start node, and an end node, while it can have an arbitrary number of properties. In most cases, relationships have quantitative properties, such as weights, costs, distances, ratings, time intervals, or strengths. Due to the efficient way relationships are stored, two nodes can share any number or type of relationships without sacrificing performance. Although they are stored in a specific direction, relationships can always be navigated efficiently in either direction

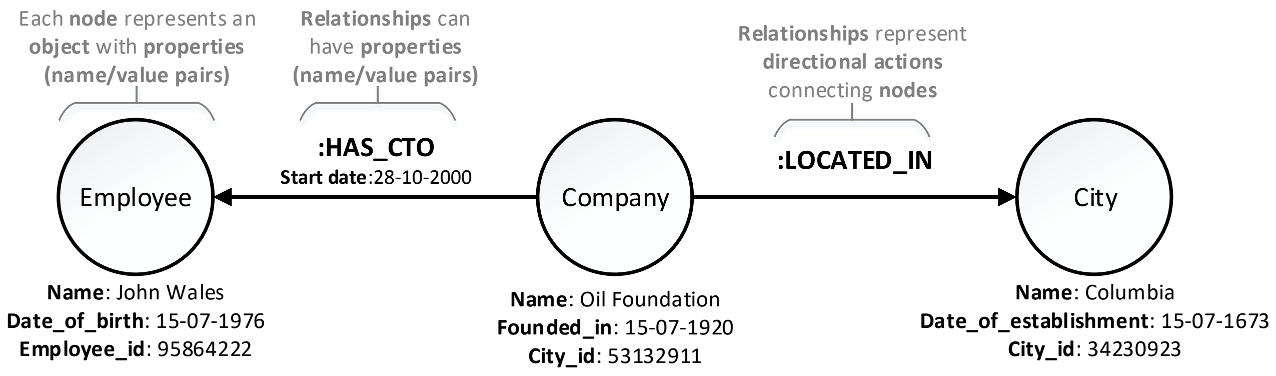

Each node represents an object with properties (name/value pairs)
Relationships can have properties (name/value pairs)
Relationships represent directional actions connecting nodes
:HAS_CTO
Start date:28-10-2000
:LOCATED_IN
Employee
Company
City
Name: John Wales
Date_of_birth: 15-07-1976
Employee_id: 95864222
Name: Oil Foundation
Founded_in: 15-07-1920
City_id: 53132911
Name: Columbia
Date_of_establishment: 15-07-1673
City_id: 34230923

Figure 3.11: *Building blocks of the property graph model*

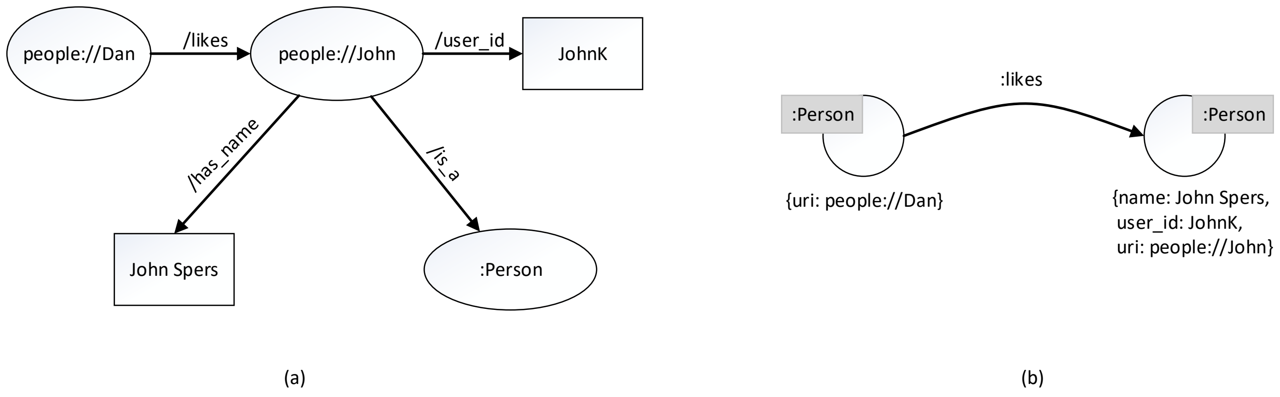


Figure 3.12: *Example of a graph representation through (a) RDF and (b) property graph*

Properties express non-relational information about nodes and relationships. In general, properties include a vertex having a name, an age and an edge having a timestamp and/or a weight. An example of a property graph is depicted in Figure 3.11. Neo4j [148], GraphDB [149] and AWS Neptune [150] are some of the available graph databases, which are based on directed and multi-relational property graphs. Nodes and edges are comprised of embedded key-value pairs, which can be defined in a schema, whereby the expression of more complex constraints can be described thoroughly.

The graph representation is a key component in a heterogeneous set of objects (nodes) that can be related to one another in a multitude of ways (edges). In a graph, each node is seen as an atomic entity that can be linked to any other node or have properties added or removed at will. This provides a more abstract view of a ”row in a table” and empowers the attitude of thinking, in terms of actors, within a world of complex relations as opposed to, in relational databases, statically-typed tables joined in aggregate. Once a domain is modelled, that model must then be exploited in order to yield novel and differentiating information. Graph computing has a rich history that includes not only query languages devoid of table-join semantics, but also algorithms that support complex reasoning: path analysis, vertex clustering and ranking, subgraph identification, and more. The world of applied graph computing offers a flexible, intuitive data structure along with a host of algorithms able to effectively leverage that structure.

Though, property graphs structures are distinguishable from the *Resource Description Framework* (RDF)

Table 3.3: *Comparison of property and RDF graphs*

Property Graph	**RDF Graph**
Swift association of properties with edges and nodes	Verbose association of structure
Community-driven	W3C and Open Geospatial Consortium (OGC) standards
Deficiency in processing multiple property graphs	Immediate handling of multiple RDF graphs simultaneously
No standard terms, vocabularies	Various curated terms, ontologies
Simpler approach with absence of formal theoretical foundation, semantics and inference	Formal theoretical foundation with interpretation, entailment and description logic

databases, which focus on querying and analyzing *RDF triples*, namely the $\{subject, predicate, object\}$ statements [151, 152]. Initially, the RDF is a family of World Wide Web Consortium (W3C) specifications, originally designed as a metadata data model [153]. The entire set of RDF triples can be represented as directed multi-relational graph, RDF databases are rendered as a special subset of graph databases. In Table 3.3, a brief comparison of property and RDF graphs is depicted. A paramount difference in graph and RDF databases is that RDF are not variable in adding additional key-value pairs to the stored graph. As depicted in Figure 3.12, RDF represents each property through a corresponding relationship, whereas the property graph includes more complex structures. Consequently, RDF databases perform adequate at representing complex metadata, reference, and master data, along with the representation of concepts with high complexity, where semantic context and data quality are critical [154]. The usage of the RDF framework enables the execution of substantially more complex and expensive queries, along with schema, compared to the property frameworks [155].

3.4.3 Concepts and characteristics of NoSQL Databases

One of the main characteristics of the NoSQL data stores is the effective and horizontal scalability with the addition/removal of servers into the resource pool. Although, there have been attempts to scale relational databases horizontally, on the contrary, RDBMSs are designed to scale vertically in terms of adding/removing more power to a single existing server. With regard to what is being scaled, three scaling dimensions are considered, namely the scaling read requests, write requests, or data storage. The partitioning, replication, consistency, and concurrency control strategies used by the NoSQL data stores have significant impact on their scalability. More specifically, partitioning determines the distribution of data among multiple servers and a means of achieving all three scaling dimensions. Another important factor in scaling requests is replication as storage of redundant data on multiple servers delivers efficient distribution over them. Replication also comprises a key part in providing fault tolerance as data availability can withstand the single point failure of one or more servers. Furthermore, the choice of replication model is also highly correlated to the consistency level provided by the data store. Additionally, an influential factor in scaling requests is concurrency control. Simple read/write lock techniques may not provide sufficient concurrency control for the read and write throughput required by NoSQL data stores. Therefore, most solutions use more sophisticated mechanisms, such as optimistic locking with multi-version concurrency control. In the following section, the aforementioned strategies of NoSQL data stores, along with the provided querying capabilities, are discussed and listed in Tables 3.4 and 3.5 respectively.

3.4.3.1 Consistency

As one of the CAP properties, consistency ensures that a transaction renders a database in a valid state to another, relating to how data are used and seen among the servers after each update operation. As stated below, the consistency mechanisms are highly correlated with the replication and concurrency mechanisms, due to their nature. The two main categories of consistency models are the *strong* and *eventual* consistency models. The strong or immediate consistency model provides the assurance that every write operation is visible to all subsequent read requests of the same data entity. Synchronous replication usually ensures strong consistency, but its use can be unacceptable in NoSQL data stores, in terms of increased latency. As depicted in Table 3.4, among the observed NoSQL data stores with replication, HBase exclusively supports strong consistency.

In the eventual consistency model, changes eventually propagate through the system within a period of time delay. Consequently, some server nodes may contain outdated data until their update. Asynchronous replication, should there are no other consistency mechanisms, will lead to eventual consistency as there is a lag in the propagation. As NoSQL data stores typically replicate asynchronously, and eventual consistency is often associated with them, it was expected that the reviewed NoSQL solutions provide eventual consistency. However, as depicted in Table 3.4, the majority of these data stores allow the regulation of the consistency model using alternate consistency mechanisms.

3.4.3.2 Replication

Most of the NoSQL systems maintain multiple copies of the data for availability and scalability purposes. Through database replication, the allocation of information is involved so as to qualify consistency between redundant resources, such as software or hardware components, and to deliver reliability, durability, fault-tolerance, and accessibility. The role of replication aims at the storage of the same data on multiple servers in order to provide assurance in the distribution of the read/write operations. Replication comprises a key point in providing fault tolerance as data availability can withstand the failure of one or more servers. Furthermore, the choice of replication model is also strongly related to the consistency level provided by the data store. On the context of replacing failing servers and coping with these temporary deficiencies, it is studied only full availability or full consistency can be qualified [156]. In case of an synchronous update of all replicas of a master server, a write operation is not able to be executed other than its slaves. In platforms with reliability on high availability, the technique of strong consistency is prohibitively stated as few milliseconds of latency can contribute to the loss of partitions. For example, a delay of 100 milliseconds has reduced Amazon's sales by 1% [157]. The reflection of inventory levels for the products in a consistent system will lead to an updated product catalogue for each operation. In eventual consistent system, the inventory levels may not be accurate for a query at a given time, but will eventually become accurate as data are eventually replcated across all nodes in the database cluster. Additionally, Google faced a traffic reduction of 25% as 30 instead of 10 searches increased the latency to 900 milliseconds [158]. The increase of availability will lead to consistency cut backs as the asynchronous update will enable inconsistent read operation for a short period of time. Developers are encouraged to issue idempotent queries with additional control logic in the applications. Most NoSQL systems offer atomicity guarantees at the level of an individual record. Since SQL system can bring together related data that could be modelled across separate parent-child tables, atomic single-record operations provide transaction semantics that meet the data integrity needs of the majority of applications.

Replication techniques improve system reliability, fault tolerance, and durability. Two main approaches to replication can be distinguished in two categories, namely the *master-slave* and *multi-master* replication. As shown in Figure 3.13, the master–slave model replication deploys a single node is designated as a master and is the only node that processes transactions and propagates the changes. In the multi-master replication model, multiple nodes can process write requests, which are propagated to the remaining nodes. In contrast to the master-slave replication model, the propagation happens in different and every potential connection of master entities. The design of the replication model has an immediate repercussion on the ability of the database to scale read/write requests. Master-slave replication is generally useful for scaling read requests as it allows the multiple slaves to accept read requests. However, some techniques do not permit read requests on the slave nodes, resulting solely in failover and disaster recovery. In addition, this replication model do not scale write requests because the master is the only node that processes write requests.

On the other hand, multi-master replication model is usually capable of scaling read/write requests as all nodes can handle both requests. Another replication characteristic with a great impact on data stores throughput is how write operations are propagated among nodes. Synchronization of replicas can be synchronous or asynchronous. In synchronous or eager replication, changes are propagated to replicas before the success of the write operation is acknowledged to the client. This means that synchronous replication introduces latencies because the write operation is completed only after change propagation. This approach is rarely used in NoSQL because it can result in large delays in the case of temporary loss or degradation of the connection. In asynchronous or lazy replication, the success of a write operation is acknowledged before the change has been propagated to replica nodes. This enables replication over large distances, but it may result in nodes containing inconsistent copies of data. However, performance can be greatly improved over synchronous replication.

The replication policies applied in the famous NoSQL databases are depicted in the Table 3.4. The master–slave replication policy is adopted from the Redis and HBase. On the other-hand, the multi-master replication exists in CouchDB and Couchbase Server. The data stores of Voldemort, Riak and Cassandra, support masterless replication, similar to multi-master replication as multiple nodes accept write requests. Yet, as pointed by the term masterless, all nodes play the same role in the replication system. Note that all three of the data stores with masterless replication use consistent hashing as a partitioning strategy. The strategy for placing replicas is closely related to node position on the partitioning ring.

3.4.3.3 Concurrency

In the NoSQL data stores, concurrency control is of paramount consideration as the support of a large number of concurrent users and very high rates of transaction is deemed a necessity. The definition of concurrency is the ability of multiple processes to access and change shared data at the same time. The number of concurrent

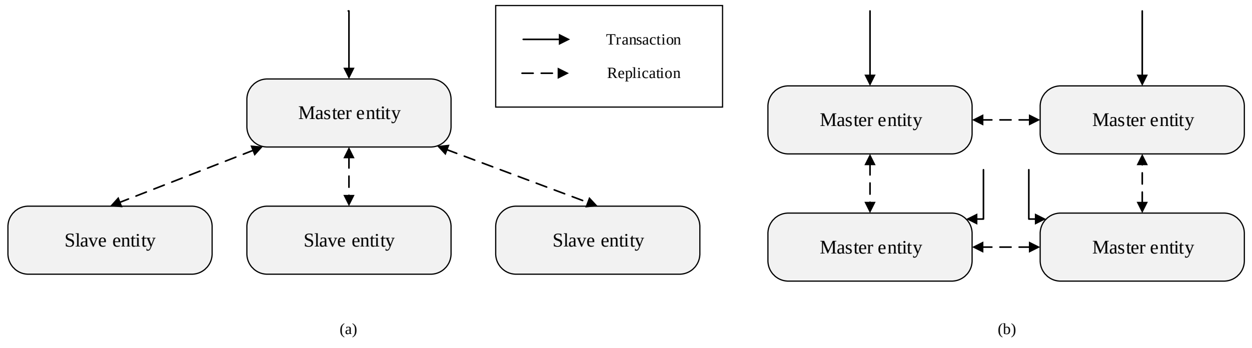


Figure 3.13: *Replication models of the (a) Master-slave and (b) Multi-master approach*

user processes that execute without blocking each other is direct proportional to the concurrency of the database system. The concurrency methods need to incorporate the control as a means of achieving simultaneous access to the same entity, row, or record on a single server node. The multiple user access of data in parallel demands strategies for surpassing inconsistencies based on the conflicting operations. In traditional databases, locking costs are low and datasets are accessed for a short period of time. On the opposite, database clusters must satisfy higher read request rates with web applications over large distances, rendering old consistency strategies as deficient.

The most popular scheme today to maximize parallelism without sacrificing serializability is the *Multi-version concurrency control* (MCC or MVCC). MVCC is a concurrency control method commonly used by database management systems to provide concurrent access to the database while it is used also in programming languages to implement transactional memory [159]. MVCC provides concurrent access to the database without locking the data, thus readers do not block writers and vice versa. MVCC delivers to each user a connection to the database with a consistent point in *time-snapshot* of the data and different versions, keeping the old values of a data item when the item is updated. Other users of the database do not see any changes until the transaction is committed. Similarly, MVCC is similar to row locking in terms of conception, without the need for page or row read locks. When a transaction requires access to an item, an appropriate version is chosen to maintain the serializability of the currently executing schedule. The notion focuses on operations that would be rejected in other techniques can still be accepted by reading an older *version* of the item to maintain serializability. When a transaction writes an item, it writes a new version and the old version(s) of the item are retained. Some multiversion concurrency control algorithms use the concept of view serializability rather than conflict serializability. On top of these, MVCC techniques present better performance on the grounds that read locking is not necessary as it does not wait to acquire lock conditions. Though, since different versions are hosted in the database, MVCC causes higher system complexity and storage space requirements.

Several other solutions are provided by the transactional isolation through locking in a transaction. The conventional approaches to transactional locking are the *optimistic locking* and *pessimistic locking* [160]. In optimistic concurrency control techniques, also known as validation/certification techniques, all updates are applied to local copies of the data items that are kept for the transaction during its execution. A record is read for the version number while an assessment is executed in order to check that the version hasn't changed before the record is written back. Several methods incorporate dates, timestamps or checksums/hashes. When the record is written back, the version is updated to make sure it's atomic and not been updated between the check of the version and the write operation of the record to the disk. This strategy is most applicable to high-volume systems and three-tier architectures, where a connection is not maintained to the database for the session. In this situation, the client connection is not actually maintained with database locks, as the connections are taken from a pool and the same connection is not the same from one access to the next. As depicted in Table 3.4, Voldemort, CouchDB, Riak, and HBase implement optimistic concurrency control with MVCC. The pessimistic locking strategy incorporates the lock of the record for the exclusive use for every operation, presenting better integrity than optimistic locking. The lock is placed onto the accessed entity so that exclusive access is guaranteed to a single operation. Every other client trying to gain access in the same data must wait until the first lock terminates. The locked entity depends on the underlying data model. More specifically, key-value stores lock records consisting of key-value pairs, column-family stores lock rows, and document stores enforce locking at document level. As depicted in Table 3.4, Neo4J locks are acquired on nodes and their relationships while MongoDB implements readers-writer locks allowing multiple readers to access data or a single writer to modify them.

3.4.3.4 Partitioning

Most NoSQL data stores implement techniques of *horizontal partitioning* (or sharding), which incorporates storing sets or rows/records into different segments (or shards) that can be located on different servers. In the contrary, *vertical partitioning* involves the storage of sets of columns into different segments and their corresponding distribution. The data model of the NoSQL store is a significant factor in defining strategies for data store partitioning. For example, vertical partitioning segments contain predefined groups of columns and data stores from the column-family category and can provide vertical partitioning in addition to horizontal partitioning. With the increase of the data volume and the surpass of the capacity of one server, partitions are a vital technique for the efficient storage in the database systems. The normalization of the data model and the ACID guarantees comprise an obstacle in the horizontal scale of relational databases, while the combination of two relational databases, instead of one, does not lead to a proportional increase of performance. This issue led to the adaptation of *web-scale* databases that are specifically designed to scale horizontally and satisfy the substantial requirements of the performance and capacity. However, NoSQL systems diverge in the data distribution technique on multiple machines. As the data models of key-value, document-oriented and column family stores are based on key retrieval, the two common partition strategies follow the same notion of keys.

Depicted in Figure 3.14i, the first partition strategy is called *range-based partitioning* and focuses on the dataset distribution through the assigned keys in each data model. A routing server is responsible for distinguishing the whole key space into blocks and allocating them into different nodes. Consequently, each node is responsible for the storage and the request handling for its relevant key block. The retrieval of a key is routed from the routing server to reach the corresponding partition. The advantage of this approach is the effective processing of range queries since adjacent keys often reside in the same node. However, this approach can result in hot spots and load-balancing issues. For example, if the data are processed in the order of their key values, the processing load will always be concentrated on a single or a few servers. Another disadvantage is that the mapping of ranges to partitions and nodes must be maintained, usually by a routing server, so that the client can be directed to the correct server. HBase, Cassandra, and MongoDB deploy range-based partitioning

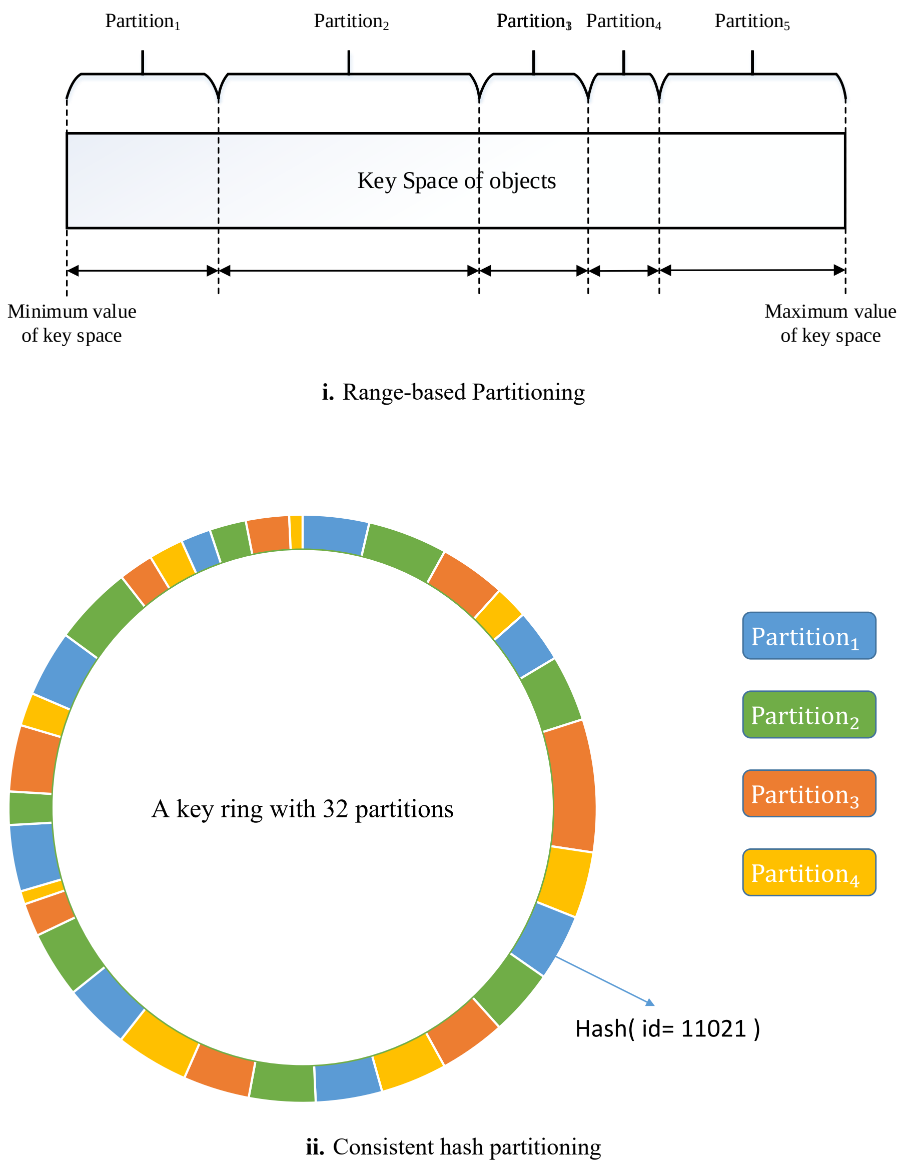


Figure 3.14: *Partitioning strategies of i. Range-based and ii. Consistent hash partitioning*

strategy as depicted in Table 3.4.

The second partition strategy is the *consistent hash partitioning*, in which keys are distributed using hash functions. Figure 3.14ii depicts the consistent hash partitioning strategy where every server is responsible for a predetermined hash spectrum. This strategy is prone to immediate executions of certain redirections of hashes. Thus, it is obvious that efficient hash functions optimally contribute keys without any need of load balancing. This type of partition strategy contributes to the dynamic cluster resizing as the addition/removal of servers does not affect the architecture and the performance. The key difference of the two approaches is denoted in the performance of the range queries. The former partition strategy delivers great performance in the execution of range queries since neighbour keys are stored in close physical server entities. Additionally, the location of an object can be calculated very fast and there is no need for a mapping service as in range partitioning. However, consistent hashing negatively impacts range queries as neighbouring keys are distributed across a number of different nodes. Although, the main advantage of range-based partitioning is the single point failure of servers since a server failure will lead to a loss of a cluster of information. Since information is grouped in servers and a specific range, a part of information stored in the corresponding server will be absent. Riak, Voldemort, Cassandra, CouchDB, DynamoDB and Clustrix deploy consistent hash partitioning strategy as depicted in Table 3.4.

In contrast to the aforementioned techniques, graph databases present different partitioning strategies due to the absence of keys. The split of a graph entity is not straightforward at all as graph information cannot be retrieved through simple key distributions. Graphs are highly mutable structures with unstable keys. The graph data can be accessed only with the exploitation of the relations among entities. Consequently, graph partitioning attempts to achieve a trade-off between two conflicting requirements. Firstly, the related graph nodes must be located on the same server to achieve good traversal performance. Secondly, the concentration of multiple graph nodes should be avoided on the same server as it will lead in heavy and concentrated load. A number of graph-partitioning algorithms have been proposed, but their adoption in practice is difficult [161]. The analysis of the relationships between the entities is the main factor in the adopted partition strategy. The several graph algorithms contribute to the identification of hotspots of strongly connected nodes in the graph entity, which can be afterwards stored in one machine. Since graphs can be rapidly altered, graph partitioning is not viable without a-priori domain-specific knowledge and many complicated techniques [162, 163]. Depicted in Table 3.4, the graph databases do not employ an internal partition strategy.

3.4.3.5 Query Capabilities

The querying capabilities of data stores comprise an paramount key aspect for a particular scenario. The diverse data stores provide different APIs and access interfaces to interact with the end users. Data models in the several approaches are highly correlated with the design of the query capabilities of the database. The analysis of the supported queries is of paramount importance in the available data model. NoSQL systems bear the notion of flexibility in their nature and incorporate it in the provided query functionalities. The current state of NoSQL spectrum is exactly right before the time of the introduction of SQL. Likewise, the majority of the heterogeneous databases is raised in the last years with several differences in the offered data model, query languages, APIs, etc, with each NoSQL store presenting its corresponding query language.

Another important query-related feature of NoSQL data stores is their level of support for MapReduce.

Table 3.4: *Partitioning, replication, consistency and concurrency mechanisms*

Partitioning , replication, consistency and concurrency mechanisms

NoSQL data stores		**Partitioning**	**Replication**	**Consistency**	**Concurrency**
Key-value stores	**Redis** [164]	No partitioning Can be implemented in the client side	Master–slave asynchronous replication	Eventual consistency Strong consistency if slave replicas are prone to failover	Optimistic or pessimistic concurrency control in application layer
	Voldemort [135]	Consistent hashing	Multi-master asynchronous replication	Configurable	MVCC with vector clock
	Riak [165]	Consistent hashing	Multi-master asynchronous replication	Configurable	MVCC with vector clock
	Memcached [166]	Clients’ implementation Most clients supportconsistent hashing	No replication Addition in application layer	Strong consistency	Optimistic or pessimistic concurrency control in application layer
Column family stores	**Cassandra** [167]	Consistent hashing and range partitioning	Multi-master asynchronous replication	Configurable	Timestamps for recent update in client side
	HBase [168]	Range partitioning	Master–slave or multi-master asynchronous replication	Strong consistency	MVCC
	Amazon DynamoDB [134]	Consistent hashing	Three-way replication across multiple zones Synchronous replication	Configurable	Optimistic or pessimistic concurrency control in application layer
	Amazon SimpleDB [?]	Manual addition of tables without internal interaction	Region definition of replicas	Configurable	Optimistic or pessimistic concurrency control in application layer

Table 3.4: *Partitioning, replication, consistency and concurrency mechanisms (continued)*

Partitioning, replication, consistency and concurrency mechanisms

NoSQL data stores		**Partitioning**	**Replication**	**Consistency**	**Concurrency**
Document-oriented stores	**CouchDB** [139]	Consistent hashing	Multi-master, asynchronous replication Offline maintanance of multiple replicas' copies of the same data and later synchronization	Eventual Consistency	MVCC
	MongoDB [138]	Range partitioning on common fields of collections	Master–slave asynchronous replication	i) Set connection read only from primary ii) Set write concern parameter to "Replica Acknowledged"	Multiple granularity locking
	Couchbase [140]	Consistent hashing with consulted table for detection of the server that hosts that bucket.	Multi-master	Strong consistency in a cluster Eventual consistency across clusters	Implementation of optimistic or pessimistic concurrency control on application layer
Graph Databases	**Neo4J** [148]	No partitioning Only cache sharding	Modified master–slave with provision of access on all server nodes Slave synchronization with propagation to master	Eventual Consistency	Locks on write operations
	HyperGraphDB [169]	Graph parts reside in several nodes Builds on autonomous agenttechnologies	Multi-master asynchronous replication	Eventual Consistency	MVCC
	AllegroGraph [170]	No partitioning	Master–slave	Eventual Consistency	100% read concurrency Near full write concurrency

MapReduce, developed by Google, is a programming model and an associated implementation for processing large datasets [171]. It has now become a widely accepted approach for performing distributed data processing on a cluster of computers. Because one of the primary goals of NoSQL data stores is to scale over a large number of computers, MapReduce has been adopted by most of them. Similarly, SQL-like querying has been a preferred choice because of its widespread use over the past few decades, and it has now also been adopted in the NoSQL world. Therefore, some of the prominent NoSQL data stores offer a SQL-like query language or similar variants. More specifically, as depicted in 3.5, some of the prominent NoSQL data stores offer an SQL-like query language, such as MongoDB, or similar variants such as CQL offered by Cassandra and SparQL by Neo4j and Allegro Graph. On the other hand, a command-line interface (CLI) is usually the simplest and most common interface that a data store can provide for interaction with itself and is therefore offered by almost all NoSQL products. In addition, most of these products offer API support for multiple languages. Moreover, a REST-based API has been very popular in the world of Web-based applications because of its simplicity. Consequently, in the NoSQL world, a REST-based interface is provided by most solutions, either directly or indirectly through third-party APIs.

Key-value stores provide simplistic APIs that provide key-based GET, DELETE and PUT operations only, due to the relevant simple data model. Any further complex query functionality has to be implemented in the application layer of abstraction, composed by the conventional query functionalities. Though, as the data model is retained in plain complexity layer, any query language would be deemed as an overhead, introducing system and performance complexity. Thus, key-value stores should be avoided if additional complexity in the query functionalities is necessary.

Contrary to the key-value stores, document-oriented databases deliver substantially richer query capabilities and APIs. Range queries on diverse values and nested documents, indexes on primary and secondary keys and combinations of plain queries with logical operators are some of the query capabilities that are available in the document-oriented databases. Via the high demand of document-oriented databases in the industry, the diverse user-friendly query languages are provided for each store, yet developers are providing custom input in its design for satisfying personalized query necessities. An attempt of unifying the diverse query candidates, the UnQL project is establishing a common query language offering an SQL-like syntax for queryimg document-oriented stores [172]. In general, UnQL can be thought of as a superset of SQL focused on collections and documents, as opposed to tables and rows. UnQL evolves on the experience of SQL language, supplementing it with syntax and concepts appropriate for the unstructured, self-describing data formats of post-modern applications. On top of that, since the demand of document-oriented databases is increasing, MapReduce applications come to integrate this notion by offering parallelizing capabilities [173].

Additionally, column-family stores also provide storage of huge amounts of semi-structured data with MapReduce frameworks been provided for the efficient processing of the parallelized calculations on a low abstraction level. In the context of column-family stores, there is a restriction only to range queries and operations, such as *in*, *and/or* and regular expressions on row keys or indexed values. The diverse column-family stores offer an SQL-like query language to provide a user-friendly interaction in the context of row keys and indexed values with where-clauses. As the query languages of column-family stores are designed individually for each store, no common query syntax exists for the subset of column-family databases.

Opposed to the aforementioned NoSQL categories, the data model of graph databases demands higher level of semantics in the query language. Graph pattern matching strategies are based on the part retrieval of the initial

Table 3.5: *Query features of NoSQL databases*

Querying capacity						
NoSQL data stores		**RESTful API**	**MapReduce capabilities**	**Query language/ Other API**	**License**	**Comments**
Key-value stores	**Redis** [164]	Third-party APIs	No	API and CLI features in diverse languages	BSD license	Server-side scripting support
	Voldemort [135]	In progress	Yes	API and CLI features in diverse languages	Apache 2.0 license	-
	Riak [165]	Yes	Yes	API and CLI features in diverse languages	Apache 2.0 license	Filtering through key filters Configurable secondary indexing Solr search capabilities
	Memcached [166]	Third-party APIs	No	API and CLI features with ASCII protocols for custom development	BSD license	No server-side scripting support
Column family stores	**Cassandra** [167]	Third-party APIs	Yes	API and CLI features with Thrift interface	Apache 2.0 license	Secondary indexing mechanisms including column families super-columns, collections
	HBase [168]	Yes	Yes	Java client API	Apache 2.0 license	Server-side scripting support with secondary indexing mechanisms
	Amazon DynamoDB [134]	Yes	Amazon Elastic MapReduce	API features in diverse languages	Closed source with pricing	Secondary indexing based on attributes other than primary keys
	Amazon SimpleDB [174]	Yes	No	Amazon API	Closed source with pricing	Automatic indexing for all columns

Table 3.5: Query features of NoSQL databases (continued)

Querying capacity

NoSQL data stores		**RESTful API**	**MapReduce capabilities**	**Query language/ Other API**	**License**	**Comments**
Document-oriented stores	**CouchDB** [139]	Yes	Yes	API features in diverse languages	Apache 2.0 license	Server-side scripting Secondary indexing support
	MongoDB [138]	Yes	Yes	API and CLI features in diverse languages	Free GNU AGPL v3.0 license	Server-side scripting Secondary indexing support
	Couchbase [140]	Yes	Yes	API and CLI features with ASCII protocols for custom development	Free and paid edition	Server-side scripting Secondary indexing support
Graph Databases	**Neo4J** [148]	Yes	No	API and CLI features in diverse languages	NTCL + (A)GPLv3	Server-side scripting Secondary indexing support
	HyperGraphDB [169]	Yes	No	Java API with Scala features	GNU LGPLv3.	Search engine and Seco scripting IDE
	AllegroGraph [170]	Yes	No	API features in diverse languages	Free, developer and enterprise versions	Solr indexing and search

graph, matching a defined graph pattern. The graph traversal starts from a chosen node and scans the graph based on the provided description. The traversal strategies are based on the basic types of graph search algorithms, namely the *depth-first* and the *breadth-first* searching algorithms. The former type of algorithm travels from a starting node to an end node before repeating the search down to a different path from the same start node. This procedure is repeated until the query is answered. Generally, depth-first search provides great results when discrete pieces of information are present as they provide an efficient strategic choice for general graph traversals. A classic/basic level of depth-first search algorithm is an *uninformed search*, where the algorithm searches a path until it reaches the end of the graph, then backtracks to the start node and tries a different path. On the contrary, dealing with semantically rich graph databases allows for *informed searches*, which conduct an early termination of a search if nodes with no compatible outgoing relationships are found. As a result, informed searches also have lower execution times. Breadth-first search algorithms conduct searches by exploring the graph one layer at a time, beginning with nodes one level deep away from the start node, followed by nodes at lower depths and so on until the entire graph has been traversed.

On top of that, the majority of the graph databases offers REST APIs, program language interfaces and specific query languages for the access of data based on the aforementioned graph search algorithms. On the contrary of the previous NoSQL categories, specific query languages are supported by the majority of the graph databases. SPARQL [175] provides declarative query functionalities with plain syntax requirements. It is supported by most RDF stores and some plain graph databases. Gremlin [176] is a low–level graph traversal language, using compact syntax based on XPATH. Cypher [177] is a declarative graph query language that allows expressive and efficient querying with the ability of updating the property graph. Depending on the query, results can be nodes with all attributes, individual attributes or aggregated data.

Chapter 4

The NFV Marketplace

The notion of the NFV/SDN is an emerging concept, leveraging commodity servers and storage, including cloud platforms, to enable rapid deployment, reconfiguration and elastic scaling of network functionalities. The biggest challenge in the customization of the 5G infrastructure through NFV/SDN is the efficient composition of the several NSs from heterogeneous networks. By any means, it is exceptionally essential to provide accessibility of users to certain portions of the core of the network for customization and optimization. Apart from the paramount requirement of laying the foundations for innovation, a unified framework is deemed indispensable for the exposure and the regulation of the services. Thus, VNFs/NSs can be published and brokered/traded by several developers, while customers can browse and select the services and virtual appliances which best match their needs, as well as negotiate and be charged under various billing models. Within the scope of the future Internet, an open marketplace comprises a key element in the advertisement and publication of the developed projects from different vendors, introducing diversity in the context of web services [178].

Albeit, the notion of a centralized component of advertising and purchasing services is in absence in the 5G infrastructure. The adoption of the open marketplace in the 5G infrastructure will expound diversity among the composed and published NS by facilitating their development and their management. The exposure of the developed NS in this framework will provide the availability of their standardization and comparison. To this effect, Service Providers will be provisioned with a plethora of choices from the various stakeholders, with seamless and dynamic deployment with real-time evaluation of the composed end-to-end NS. For the perspective of the developer, different open solutions can be contributed to each service development. Also, the open marketplace will introduce the conception of the economic plane in the exchanges of VNFs/NS via the several Service Providers. In this context, the dynamic deployment of VNFs/NS will enable the recording of economic contracts between the service providers and the Service Users [179].

In this chapter, the design of an open *NFV marketplace* as a multi–faceted repository is established, supporting the developers and TSPs with storage and publication of the developed VNFs/NS in the 5G infrastructure. Such open NFV marketplace will not only provide interoperable solutions for exchanging and trading of VNFs/NS on-the-fly, but also the centralized inclusion of information from the entire 5G ecosystem regarding resources usage, business requirements and QoS/QoE elements. Thus, this design addresses different stakeholders needs and facilitates the involvement of diverse actors in the NFV scene and attract new market entrants, through the efficient collection of all the circulated information in the 5G infrastructure. Finally, the project of the NFV Marketplace is available in [180].

4.1 Related Work

The concept of an open marketplace for publishing and standardizing services finds its roots from different domains in the framework of diverse verticals. Yet, future Internet design shares an abundant amount of goals with the 5G networks, while it has provided the basis for an open marketplace from the previous years.

Universal Plug and Play (UPnP) [181] comprised the initiation of the web service advertisement from multiple available devices. Through networking protocols, networked devices can seamlessly discover every presence on the network and establish functional web services for circulating data, communications, video transmission, etc. Such devices are Wi-Fi access points, mobile devices, firewalls, etc. Through the usage of the Simple Service Discovery Protocol (SSDP), the services declare their existence and are discovered by other devices on a network. After an application has discovered a particular device and its services on a network, it can query and modify the service's properties or call the device service's functions. UPnP provided the availability of attaching services to the control points in the network, providing the dynamic discovery of latterly connected points. Besides that the UPnP provided search utilities for exploring the plethora of the multiple hosted web services, the optimization of their lifetime cycle was selected based on the minimization of the network traffic and the delivery of high QoS levels. UPnP was initially developed for residential networks without enterprise-class devices on the roadmap as it was deemed unsuitable for deployment in business settings for economy, complexity, and consistency reasons.

One of the major notions of an open marketplace is Open Service Gateway Initiative (OSGi) industry plan, constituting one of the standard ways of maintaining web services to the Internet [182]. The OSGi standard was originally specified from the OSGi Alliance, formerly known as the Open Services Gateway initiative. The OSGi specification describes a modular system for the Java programming language that implements a complete and dynamic component model. In the form of bundles for deployment, the stored web services can be remotely managed, in terms of installation, start/stop, update and deletion, without the need of reboot. Application life-cycle management is implemented with the aim of allowing the remote downloading of management policies via interfaces. The service registry allows bundles to detect the addition of new services, or the removal of services, and adapt accordingly. The bundles provide the detailed description of the services in the service registry, while interfaces are developed for Creating, Retrieving, Updating and Deleting (CRUD) operations. Through this plan, a standard way of bridging devices, such as home appliances and security systems to the Internet. As an example, users could install a security system and modify from one monitoring service to another without having to install a new system of wires and devices.

Regardless that the open marketplaces were not initially designed for the 5G infrastructure, the previous work defined an appealing template of market structure in the NFV landscape. In the context of 5G infrastructure, an open marketplace can facilitate the high-availability storage of packaged VNFs/NS as well as a database containing the necessary meta-data of enriching the information of these entities. The ChoiceNet architecture [179] proposed a meeting ground for providing NFV advertisements and user requirements through

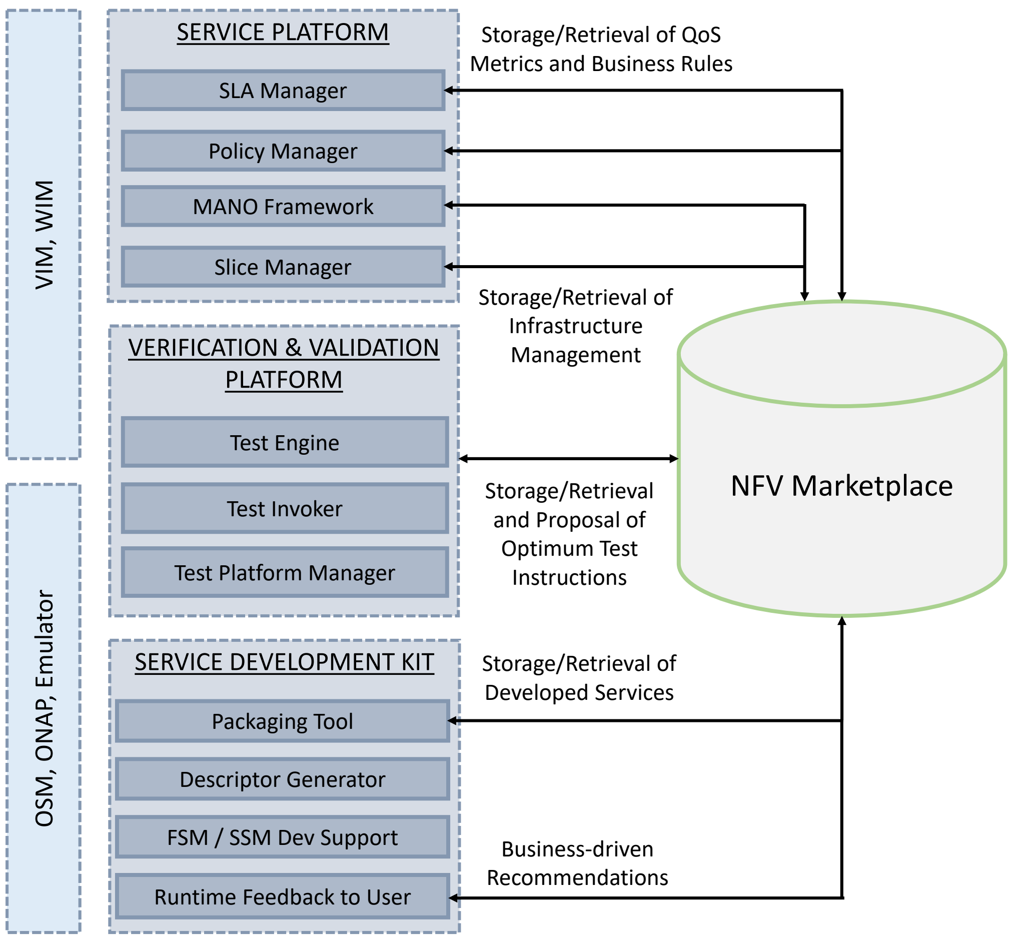

VIM, WIM
OSM, ONAP, Emulator
SERVICE PLATFORM
SLA Manager
Policy Manager
MANO Framework
Slice Manager
VERIFICATION & VALIDATION PLATFORM
Test Engine
Test Invoker
Test Platform Manager
SERVICE DEVELOPMENT KIT
Packaging Tool
Descriptor Generator
FSM / SSM Dev Support
Runtime Feedback to User
Storage/Retrieval of QoS Metrics and Business Rules
Storage/Retrieval of Infrastructure Management
Storage/Retrieval and Proposal of Optimum Test Instructions
Storage/Retrieval of Developed Services
Business-driven Recommendations
NFV Marketplace

Figure 4.1: *Positioning of the NFV Marketplace inside the 5G environment*

common minimal semantics. In the ChoiceNet framework, the deployment of several open marketplaces provide their hierarchical arrangements of offering service bundling and auction services. This proposed approach was structured with the aim of the utmost importance of economic perspective of the services, yet ignoring the integration with the production environment of the 5G infrastructure.

On top of this marketplace template, NetEx [183] constitutes a network marketplace extending to the physical layer for a cloud datacenter. This proposed approach provided the ground for the Service Providers to deploy their services in the infrastructure and the relevant set prices, requirements and policies. NetEx had the aim of motivating providers to expose necessary information for offering and, correspondingly, receiving accountable services. Although, it comprised ignoring the integration with the production environment of the 5G infrastructure.

4.2 Architecture Overview

The open NFV Marketplace constitutes a vital component in a 5G environment, implementing a multi–faceted repository for the various included folds of the infrastructure and the entire lifecycle of the development of the services. This section provides the architectural descriptions and the role of the NFV Marketplace in a typical 5G environment, in terms of interacting software components, and covers with meticulous details the architecture of the component.

4.2.1 Positioning of the NFV Marketplace in the 5G Infrastructure

In Figure **??**, a typical structure model of a dynamic 5G ecosystem is depicted, along with the three typical large subsystems that cover a holistic DevOps lifecycle of a NS. More specifically, they provide from the development (Service Development Kit, SDK), the verification and the validation (Verification & Validation Platform, V&V) up to the operation and the instantiation stage (Service Platform, SP) of the NSs. Although the functionality of each system may overlap, they typically operate at different timescales and are performed by different actors. Yet, not every environment is obliged to incorporate these systems in order to exploit the NFV Marketplace, but the qualification of their conventional and principle stages ensure the evolvability, reusability, maintainability and stability of the developed services of the 5G platform. In more detail, each interaction of the NFV Marketplace is meticulously analyzed below.

Service Development Kit (SDK): The SDK aims at providing the developer with a powerful set of tools to develop, examine and evaluate NFV-based NSs. Thus, it includes tools and mechanisms for the unified development of the NSs, namely from their creation and validation, until their final packaging and advertisement to the NFV Marketplace. Thus, the NFV Marketplace comprises the key storage point of publishing the emerging services directly from the developer. In a parallel manner, the developers have access to monitor and examine the published web services from other developers in the NFV Marketplace. Consequently, they conceptualize the idea of exploiting existed web services, or even getting inspired from the developments. Developers can rapidly initiate projects of NSs on–the–fly, either by fetching already-advertised NSs from the NFV Marketplace, or by commencing new developments. The ability of optionally enabling easy re-use of existing components comprises a crucial advantage, which can be blended into the overall service under creation [184, 185].

Service Platform (SP): In the SP, one of the core components of the 5G ecosystem is the flexible Management and Orchestration (MANO) framework [186], responsible for the orchestration features, instantiation and the management of VNFs/NSs. Along with the MANO, several other key enablers coexist as introduced in the 5G environment, such as the Slice [187, 188], Policy [189, 190] and the SLA management [188, 191]. The components of the SP produce an abounding amount of information for each respective NS. The produced information of the SLA and the Policy Manager is pertain to the negotiated QoS metrics and the business rules of the end–user with the Service Providers, through the SLA Agreements and the Policies respectively. Similarly, the MANO framework and the Slice Manager export relative information with resource allocation of the instantiated VNFs and the multiple virtual networks across the infrastructure.

The positioning of the NFV Marketplace comprises the centralized storage entity of all the components of the SP platform. On the first level, the main interactions of the SP platform with the NFV Marketplace are based on the deployment of the VNFs/NSs. Through the development process of the SDK platform, the Network Services are published to the NFV Marketplace. In order for the Network Services to be instantiated, the retrieval from the NFV Marketplace is necessary from the MANO framework, along with any relevant file deemed necessary for the deployment. Secondly, the information of the construction of multiple slices on top of the common shared physical infrastructure is stored in the NFV Marketplace by the Slice Manager. Finally, the NFV Marketplace is responsible for the storage of the information produced from the SLA and the Policy manager. After the negotiation of the Service Provider with the end-user, high level business-oriented SLA templates and the lower level technical-oriented policies are included in the signed contract. The signed contract of this negotiation contains a plethora of information, correlated with the corresponding Network Service. Thus, this fact allows the efficient design of the enrichment of the metadata of the NS in the NFV Marketplace.

Verification & Validation Framework (V&V): The V&V is an innovative ecosystem accountable for the validation and verification of the developed network services [192, 193]. Via the V&V platform, the developer ensures the quality and the reliability of their produced designs prior to release. From the perspective of the TSP, V&V stands for the validation of the designs prior to deployment on his core network. The advent of the NFV Marketplace constitutes a paramount lever in the V&V ecosystem. The main interactions of the NFV Marketplace with the V&V platform lie in the execution of the several tests and the storage of the results. After the development and the submission of the Network Services to the NFV Marketplace, the end–user is able to design tests for every available Network Service. Thus, the V&V environment retrieves the requested NSs and the corresponding tests from the NFV Marketplace. It is obvious that the test process demands a plethora of information from low level metrics, logging, system reporting, all of which are included in the NFV Marketplace. The inclusion of the concrete information in the NFV Marketplace provides a holistic insight to the end–user, even in Root Cause Analysis (RCA) scenarios.

In this ecosystem, the NFV Marketplace addresses the storage necessities of the produced entities from the several stakeholders. All of the three systems interact with the NFV Marketplace, each of which has the aim of storing and assuring the validity of its produced information. Although, in case of specialized configuration aspects such as control of visibility or security types of data stored, the local deployment of the VNF Marketplace is available. Thus, custom configurations of the local NFV Marketplace are available for each platform/stakeholder with the exact associated functionality compared to the public instance.

4.2.2 Consistent Management of Container Artifacts

One of the central concepts of the NFV landscape constitutes the circulation technique of the multiple VNFs/NSs in the infrastructure, along with interchangeable modules. After their composure and testing through the SDK, the exchange of VNFs/NSs between the diverse stakeholders, domains and SPs is achieved by the concept of *packages* [194, 195]. The advent of the notion of packaging allows an interoperable medium for sharing and exchanging container artifacts with ease of reuse. To leverage the benefits of a consistent package management, current cloud computing environments adopted this concept. The two famous examples are the prominent TOSCA Cloud Service Archive (CSAR) format [195, 196] and European Telecommunications Standards Institute Industry Specification Group for NFV (ETSI ISG NFV) [194]. The cloud package concepts are naturally very close to the NFV domain and are already in use by several research projects [197–201].

Regardless of the diverse proposed specifications, the requirements of a package are deemed predefined from the ETSI specification [194]. In Figure 4.2, a generalized model of a package is depicted, where the deployment templates of descriptors for the Network Service, the VNFs and the corresponding tests are included in the as Network Service Descriptors (NSDs) and Virtualized Network Function descriptors (VNFDs) respectively. These incorporate the description of the relevant aspects of network services and constituent VNFs in a machine-readable and structured way. The metadata of the package are included in the MANIFEST file, while

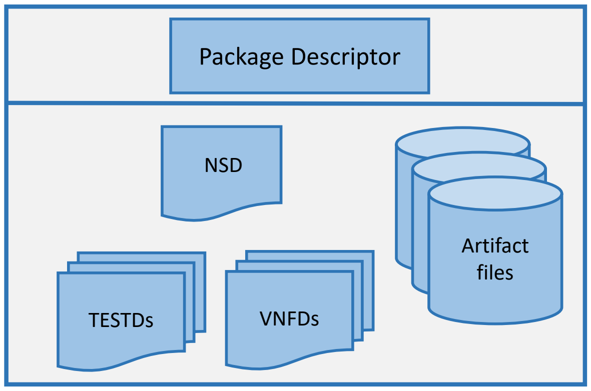


Figure 4.2: *Generalized model of the NFV package format*

the Package Descriptor (PD) acts as an index to describe the files contained in the package content. Additionally, the package bears the appropriate files as artifacts, such as configuration files, software code, and virtual machine images can be included with the aim of providing simplified life-cycle management in current cloud computing technologies.

The NS programming concepts have more or less converged in the state of the art, as it can be witnessed in the range of package formats currently available. Consequently, there is a plethora of proposed schemas of packages and descriptors with different requirements. Until now, the existing storage solutions require an a-priori knowledge of the schema of the package and the incorporated descriptors. Thus, the several solutions are confined to the package format and comprise an obstacle to the interoperable exchange of information to third party 5G platforms. The format dependency motivates the definition of a storage proposal, surpassing this obstacle of structure–dependencies. More specifically, the proposed VNF Marketplace points beyond the package dependency problem by adopting the technology of NoSQL Databases [202]. Through a NoSQL Database System, an appealing flexibility of handling structure-agnostic package/descriptor storage is feasible while the cost of schema updates is eliminated [203, 204]. Thus, the adoption of cutting-edge advances in the area of database management systems delivers performance with high availability.

The approach incorporates the proper selection of one of the most famous databases from the family of the Ducoment-oriented stores, *MongoDB* [138]. The aim of selecting MongoDB for our approach is the convenient management of complex-data document and the developer-friendly environment for interpretable JSON/JSON–like data types. Furthermore, MongoDB introduces considerably enhanced performance in the time complexity of the available CRUD operations, along with advanced query engine and index structures [205]. Since the enormous file sizes of artifact files demand paramount time durations for the circulation in the 5G infrastructure, an efficient management solution is deemed necessary. On top of that, MongoDB satisfies the necessities of storing the several artifact files, deemed critical for the service instantiation, with the incorporation of MongoDB's *GridFS* [138]. Through GridFS, the persistent storage of large binary files is injected through chunk-based structure in an instance of MongoDB.

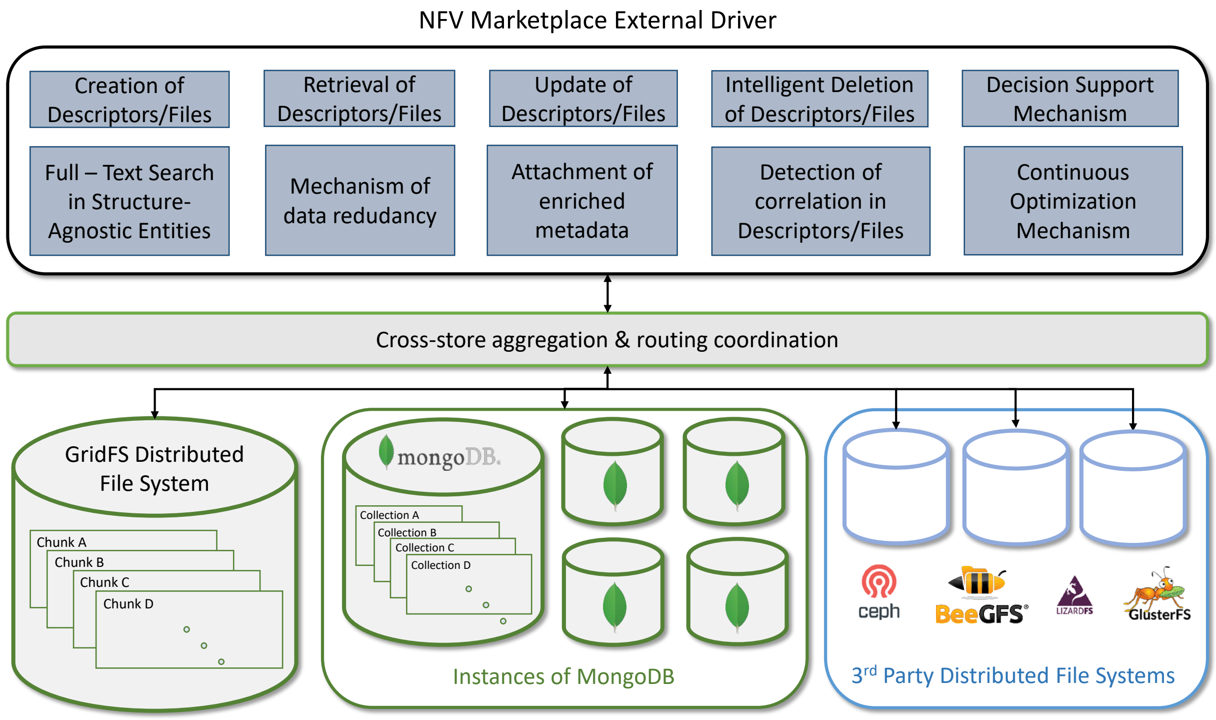

NFV Marketplace External Driver
Creation of Descriptors/Files
Retrieval of Descriptors/Files
Update of Descriptors/Files
Intelligent Deletion of Descriptors/Files
Decision Support Mechanism
Full – Text Search in Structure-Agnostic Entities
Mechanism of data redudancy
Attachment of enriched metadata
Detection of correlation in Descriptors/Files
Continuous Optimization Mechanism
Cross-store aggregation & routing coordination
GridFS Distributed File System
Chunk A
Chunk B
Chunk C
Chunk D
mongoDB
Collection A
Collection B
Collection C
Collection D
Instances of MongoDB
ceph
BeeGFS
LIZARDFS
GlusterFS
3rd Party Distributed File Systems

Figure 4.3: *Architecture and Internal Modules of the NFV Marketplace*

4.2.3 Internal Modules and Architecture of the NFV Marketplace

The NFV Marketplace is considered to be a multi-sided catalogue, addressing different stakeholder needs in the entire lifecycle of the NS, e.g., from its development in storing the design to its deployment and verification. The inclusion of the correlated information pertain to the NS enables the annotation of VNFs/NS with metadata. The attached information contribute to the enhancement of the key functions and interfaces of the NFV Marketplace for storing, searching and retrieving VNFs/NS based on these metadata, as well as providing added value services.

In Figure 4.3, the architecture of the NFV Marketplace is depicted. The deployment of a NoSQL MongoDB Database addresses the management and storage necessities of the template files of NSDs, VNFDs and PDs. The stored descriptors are required for the selection, development and instantiation of the NS by the NFV Orchestrator (NFVO) as well as for data analysis in order for the added value functionalities of NFV Marketplace to operate properly. For further enhancement, the proposal constitutes a holistic approach of the entire 5G infrastructure beyond the plain storage entity of VNFs/NS. Additionally to the storage of the descriptors, MongoDB addresses the storage necessities for every component in the 5G ecosystem, such as the SLA Manager, Policy Manager and Slice Manager, along with the V&V platform. Thus, the following object categories are defined:

- **Network Service Stored Objects**: Feasibility of storage information about all the on-boarded NSs, aiding the creation and management of the NS deployment templates.
- **Virtual Network Function Stored Objects**: Feasibility of storage information about VNFs, referring to

the description template, reference to the software images (or even the software image itself) and manifest files.

- **Package Stored Objects**: Feasibility of storage information about packages, acting as an index to describe the files contained in the package.
- **Test Stored Objects**: Feasibility of storage information about all of the constituent tests for a relevant NS, referring to the setup of the test execution environment, the preparation of the test environment etc.
- **SLA Stored Objects**: Feasibility of storage information about SLA contracts, incorporating high-level business metrics, as distinguished objectives and quality attributes.
- **Policy Stored Objects**: Feasibility of storage information about policies, including a set of enforcement rules, as the translation of the high-level business metrics.

The systematic handling of the artifact files forms an inevitable issue in the 5G infrastructure. Along with the aforementioned stored entities in the MongoDB, NFV Marketplace delivers the feasibility of storage information about artifact files through GridFS. The assets from the deployment of the MongoDB's GridFS renders it as a sufficient storage component for hosting the Artifact Stored Objects, along with the constituent package. Such artifact files are necessary components for the instantiation of VNFs/NS, such as VM disk images, early initialization of cloud instances, references to external locations, etc. The distributed environment of GridFS bears the significant advantage of laying the common file storage infrastructure for multiple potent instances of MongoDBs with optimum consistency in the hosted files. In addition to the standard GridFS standard, the NFV Marketplace supports the facilitation of third–party file systems, such as Ceph [206], GlusterFS [207], and others [208–210]. The detection of the stored files in the multiple file systems is feasible through the inclusion of reference files for external storage locations in the packages. The references are exploited by the *Coordination Router*, responsible for the management of requests and the access rights among the different deployments of the NFV Marketplace.

4.3 NFV Marketplace External Driver

The application layer of the NFV Marketplace is introduced in the *NFV Marketplace External Driver*, responsible for the management of the storage in the lower level of the architecture and the delivery of the cutting-edge functionalities. In parallel, it introduces an interface with key functionalities to the user with abstraction of the internal details of the architecture. Below, its key functions and interfaces are meticulously analyzed.

4.3.1 User Interfaces for NFV Descriptor and Package Management

In the domain of persistence storage, the CRUD functionalities comprise major requirements of any existing database. The advent of the NFV Marketplace External Driver delivers interfaces for storing, retrieving, updating and deleting files in the available databases. The exposure of the CRUD operations is enabled through RESTful APIs with the aim of providing flexibility in the 5G ecosystem, currently depicted in Table 4.1. Figure 4.4 depicts the sequential steps of the CRUD operations in the NFV Marketplace. A special operation in the

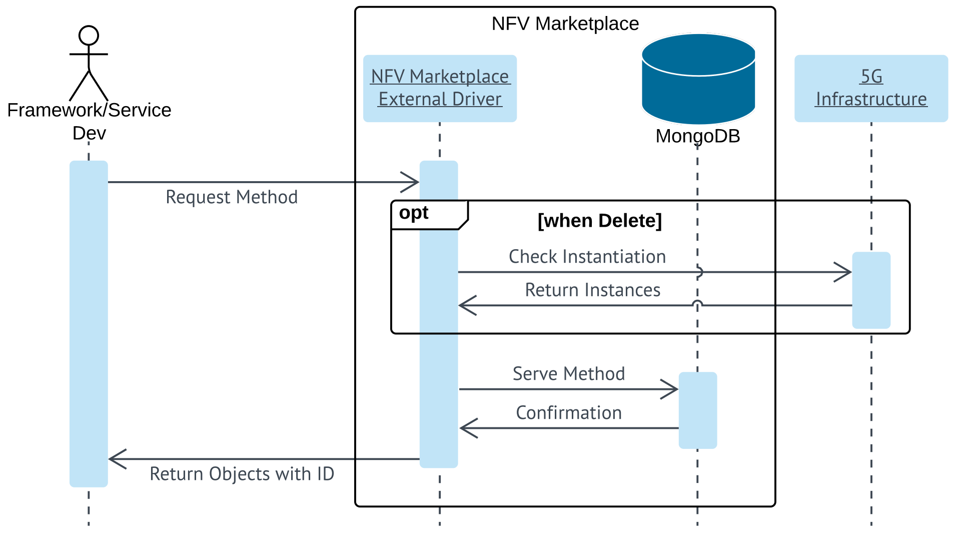


Figure 4.4: *Document management flow diagram in the NFV Marketplace*

document management is the deletion of a document where additional inspection is essential in order to ensure that a service is not instantiated in the 5G infrastructure.

4.3.2 Attachment of Enriched Metadata

The role of metadata is considered as essential in the NFV Marketplace, rendering it as an information-driven repository. The interaction of the NFV Marketplace with several components, such as the V&V platform and the SDK, offers the key advantage of collecting valuable information and being deployment–pattern aware. The functionality of the NFV Marketplace is utterly based on the metadata of all stored objects as it allows the enrichment of VNFs/NS with meaningful information, appending high–level data abstraction from the real–time operation of the 5G Infrastructure.

The role of metadata is considered as essential in the NFV Marketplace, rendering it as an information-driven repository. The interaction of the NFV Marketplace with several components, such as the V&V platform and the SDK, offers the key advantage of collecting valuable information and being deployment–pattern aware. The functionality of the NFV Marketplace is heavily and utterly based on the metadata for all stored objects. Figure 4.5 illustrates the attached metadata that are attached initially in the root level of the final JSON object, including the uploaded document in an object field and stored in the MongoDB. Through the initial upload of every entity, the NFV Marketplace differentiates it through the attachment of a Unique Universal Identifier (UUID), along with the md5 checksum for data integrity and various other information for the owner of the service. Afterwards, the NFV Marketplace enriches the record with meaningful information, appending high–level data abstraction from the real–time operation of the 5G Infrastructure. Hence, this functionality enables a plethora of services in the NFV Marketplace such as the efficient full–text search, correlation between different stored objects, perform of complex queries, versioning and other services.

Table 4.1: *Exposed endpoints for the Catalogues*

Action	HTTP Method	Endpoint
List all the available descriptors	GET	/api/catalogues/{collection}
List all descriptors matching a specific filter(s)	GET	/api/catalogues/{collection}?{attributeName}={value}[&{attributeName}={value}]
List only the last version for all descriptors	GET	/api/catalogues/v2/collection?version=last
List a descriptor using the UUID	GET	/api/catalogues/v2/{collection}/{uuid}
Store a descriptor in the Catalogue	POST	/api/catalogues/v2/collection
Update a descriptor matching a specific filter(s)	PUT	/api/catalogues/v2/collection?{attributeName}={value}[&{attributeName}={value}]
Update a descriptor using the UUID	PUT	/api/catalogues/v2/collection/uuid
Set a specific field of a descriptor	PUT	/api/catalogues/v2/collection/uuid?{attributeName}={value}
Delete a descriptor matching a specific filter(s)	DELETE	/api/catalogues/v2/collection?{attributeName}={value}[&{attributeName}={value}]
Delete a descriptor using the UUID	DELETE	/api/catalogues/v2/collection/{uuid}

```
{
"id": {
    "description": "Universally Unique Identifier",
    "type": "string"},
"document": {
    "description": "The uploaded document",
    "type": "object"},
“platform": {
    "description": “Name of platform",
    "type": “string"},
“pkg_ref": {
    "description": “Number of package references",
    "type": “integer"},
"status": {
    "description": "Availability state in the NFV Marketplace",
    "type": "string"},
"signature": {
    "description": "Signature of the docuement's developer",
    "type": "string"},
"md5": {
    "description": "MD5 checksum",
    "type": "string"},
"username": {
    "description": "The username of the document's developer",
    "type": "string"},
"created_at": {
    "description": "The creation date of the document",
    "type": "string"},
"updated_at": {
    "description": "The updation date of the document",
    "type": "string"}
}
```

Figure 4.5: *Schema of the initial attached metadata at the root level of each document*

4.3.3 Full–text Search in Structure–Agnostic NFV Descriptors and Packages

The diverse metadata have a great impact on the NSs/VNFs functionality and their respective quality provisioning. Given that the NFV Marketplace includes enriched information of the stored VNFs/NS in the 5G architecture, mechanisms are demanded to exploit it with an efficient, expeditious and analytic retrieval. In this perspective, we designed a full–text search mechanism of descriptors of the NFV Marketplace with the aim of accessing any desirable value included in the deep hierarchically-structured data. Since the enriched information of each VNF/NS is gathered in the NFV Marketplace, the machine–readable format of each file needs to be entirely transparent and utterly accessed in any detail. The technique of implementing and utilizing the full–text search functionality is based on the design of a index data structure, called *inverted file* (IF). The purpose of the IF is the storage of mappings from the content of the descriptors to its locations in the database, at a cost of increased processing when the upload occurs. In this case, the descriptors are scanned through their upload and its content is stored. Figure 4.6 depicts an example of the composition of the IF from two JSON document files. The NFV Marketplace External Driver scans the uploaded document and extracts any included field, attaching them in the IF along with the corresponding value and the ID of the document. In case of any mutual field in the uploaded documents, the values are grouped in the same field, as depicted in Figure 4.6 for the *vnf_id* field.

Generally, the persistent storage of the IF is enabled in the MongoDB, where it hosts the desirable pairs of attribute/value (defined by the developer) along with a list of unique IDs of documents for the corresponding pair. The inverted file is updated for the PUT, POST and DELETE methods, where a document is added to (or deleted from) the NFV Marketplace, and correspondingly a specific value and/or a list of IDs is updated. Figure

Table 4.2: *Syntax of the URI string for querying the NFV Marketplace*

Name	**URI String**
Filter_Expression	attributeName.op=value[,value]
Combination of expressions	Filter_Expression[&Filter_Expression]

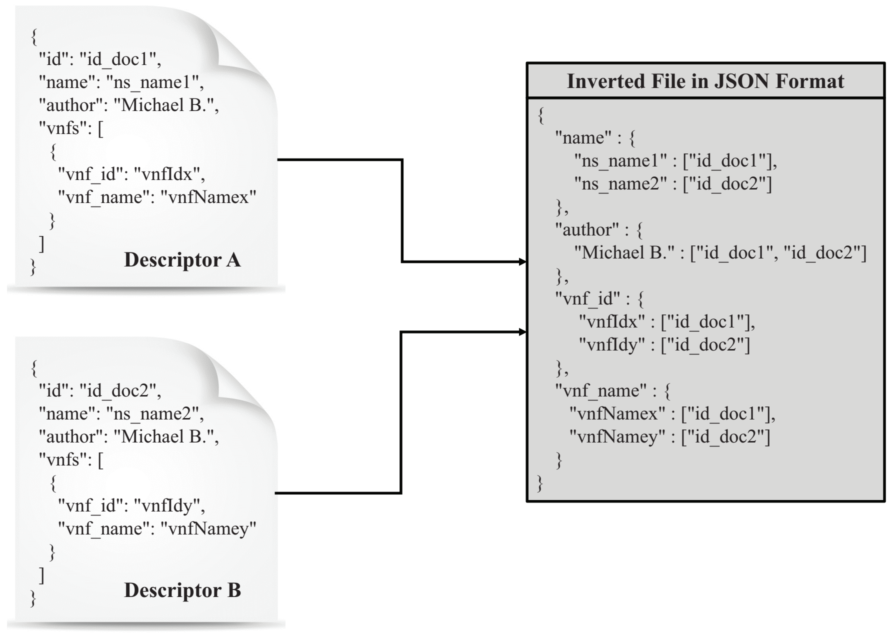
4.6 depicts the upload and search operation of documents, focusing on the update of the IF and the respective execution of search queries. In the case of the upload operation, the IF is updated with the pairs of the document, as the example in Figure 4.6, with its further upload. The search operation is triggered from the query execution to the NFV Marketplace, where the NFV Marketplace External Driver searches the IF for the matching criteria in the MongoDB. The exploration of the IF for the matched criteria of the query results in a group of document IDs, including the corresponding fields. Consequently, the NFV Marketplace External Driver retrieves the documents via the returned IDs from the MongoDB with their further exposure to the request.

Thus, the NFV Marketplace is seamlessly available to execute complex queries for every level of the machine–readable format of the descriptors. Via the searching interface, the NFV Marketplace retrieves the descriptors according to the match of the query criteria given. The ability to search NFV Marketplace is a vital prerequisite on the grounds that the number of objects stored is extortionate. Hence, the retrieval of services with particular characteristics is enabled the through the execution of queries and filtering with the corresponding criteria. The matching criteria could be from plain queries, i.e fetching a descriptor with one field, to more complex queries, i.e applying comparison query operators or combinations of criteria.

Table 4.2 depicts the designed structure of the URI string for querying and filtering the content of the NFV Marketplace, with [.] denoting optional parameters given in the URI string, *value* a scalar value and *op* the comparison query operators. The execution of complex queries is enabled through the concatenation of multiple expressions. A list of the available comparison query operators is depicted in Table 4.3. Hence, key-based filtering is available by setting an appropriate sequence of URI query parameters. The key-based filtering is applied to a GET method with the aim of returning the objects that match the particular filters.

4.3.4 Management of data redundancy

The storage capacity and its management in the application layer is of paramount importance in the 5G environment. A big challenge in the persistent storage and efficient data quality is the *excessive data redundancy*, where documents or files are duplicated, due to deficient coding [211]. In this approach, focus is concentrated in preventing the use case of duplicate documents in the NFV Marketplace through the development of a reduction management mechanism. The method of detecting duplicate records is based entirely on the attached metadata of the NFV Marketplace. The usage of an identifier would propose several issues in the retrieval of the objects in the level of the semantics [212]. Consequently, the inspection for the duplicate documents entails the search through hashing algorithms in the MongoDB. Yet, in this approach, the key factor of this mechanism is the efficient storage of the semantics in the level of the order of upload. Thus, a specific field is dedicated in



```
{
 "id": "id_doc1",
 "name": "ns_name1",
 "author": "Michael B.",
 "vnfs": [
   {
    "vnf_id": "vnfIdx",
    "vnf_name": "vnfNamex"
   }
 ]
}
```

```
{
 "id": "id_doc2",
 "name": "ns_name2",
 "author": "Michael B.",
 "vnfs": [
   {
    "vnf_id": "vnfIdy",
    "vnf_name": "vnfNamey"
   }
 ]
}
```

```
{
  "name" : {
     "ns_name1" : ["id_doc1"],
     "ns_name2" : ["id_doc2"]
  },
  "author" : {
     "Michael B." : ["id_doc1", "id_doc2"]
  },
  "vnf_id" : {
      "vnfIdx" : ["id_doc1"],
      "vnfIdy" : ["id_doc2"]
  },
  "vnf_name" : {
    "vnfNamex" : ["id_doc1"],
    "vnfNamey" : ["id_doc2"]
  }
}
```

Figure 4.6: *Composition of the inverted file*

the root level of the metadata with the aim of denoting the existence of duplicate files. Figure 4.5 depicts the *pkg_ref* field, responsible for counting the duplicacy of the document/file. By generalizing this design, various packages can contain mutual files, such as configuration scripts. The reference of a file as a duplicate with the exact number of its multiplicity will ease the functionality of the NFV Marketplace and provide a consistent storage solution.

4.3.5 Decision Support Mechanism

Aligned with the escalating development of the NFV landscape, the number of services is increasingly overwhelming in the NFV Marketplace. The publishing of the services from the various stakeholders yields a plethora of VNFs/NS selections to the end-user in the development phase. Therefore, the infiltration, prioritization and delivery of optimum information about VNF selections is considered necessary. The process of optimum selection of VNFs/NS by the operators is no exception and there is a clear need to facilitate their decision making process by providing recommendations that match their preferences and needs. Regardless of the provided full-text search utilities, the NFV Marketplace aims to comprise an active component beyond a plain data store in his interactions. The criteria of providing optimum VNF recommendations are 1. the preferences of VNF QoS/QoE requirements and VNF performance indicators of the developer 2. the VNF preferences of other developers in the infrastructure.

The former is provided through the inclusion of stored information of VNFs and the corresponding QoS/QoE requirements and performance indicators produced from the different components, such as monitoring and test information, SLAs etc. Through this information, the VNFs are characterized by a set of discrete attributes and features within the NFV Marketplace. Based on these, their analysis reveal the correlation and the optimum

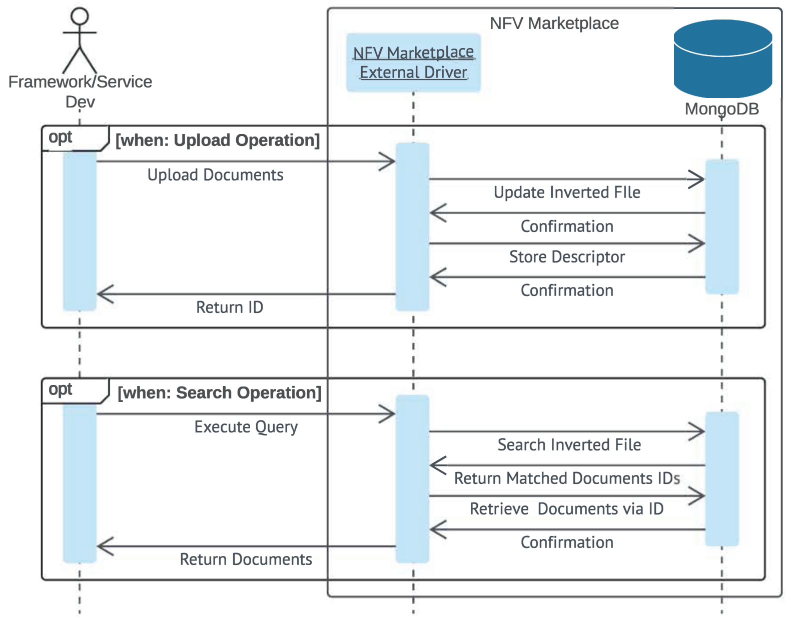


Figure 4.7: *Management of the IF and full-text search mechanism in the NFV Marketplace*

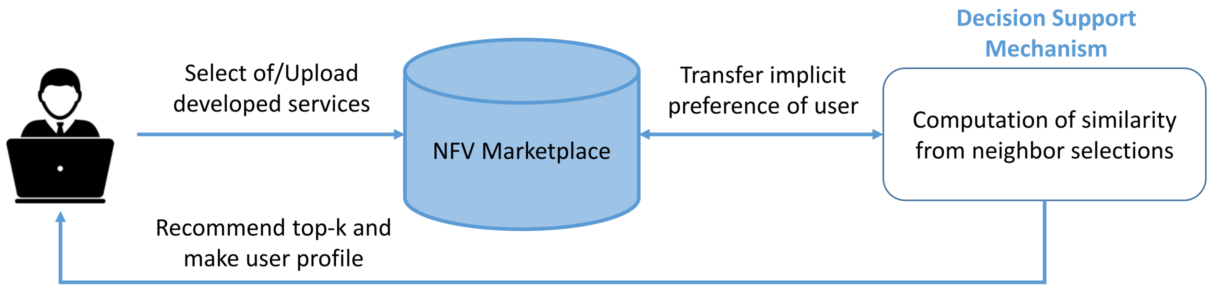

Select of/Upload developed services
NFV Marketplace
Transfer implicit preference of user
Decision Support Mechanism
Computation of similarity from neighbor selections
Recommend top-k and make user profile

Figure 4.8: *Incremental Item-to-Item Collaborative Filtering in the NFV Marketplace*

similarity of the available VNFs with the selected requirements. The second criterion is the consolidation of the forwarding graph of the NS, revealing implicit preferences of VNF selections, relative to the way of connecting such entities. Based on these two criteria, the NFV Marketplace targets in detecting qualitative similarities with the VNF requirements of the developer, going beyond to recommending VNF options. This approach is based on a well-known family of recommender systems, namely the *collaborative filtering* approach, where the developer receives suggestions on items that are similar with his past selections. In Figure 4.8, the implemented recommendation system is depicted, integrated with the NFV Marketplace. The developed recommendation principle is called *Incremental Item-to-Item Collaborative Filtering* as it aims in training in each sequential sample inserted in the mechanism [213]. As soon as the developed services are uploaded in the NFV Marketplace, the Decision Support mechanism is seamlessly notified on training with the new hosted sample. The incremental approach avoids the re-training with the entire subset of entities and delivers immediate recommendations to the user. The implementation of the Decision Support mechanism is available in [214].

Table 4.3: *Available operators implemented for querying the NFV Marketplace*

Comparison Operator (op)	**Description**
attributeName.neq=value	Rightmost attribute is not equal to the value
attributeName[.eq]=value	Rightmost attribute is equal to the value
attributeName.gt=value	Rightmost attribute is greater than the value
attributeName.gte=value	Rightmost attribute is greater than or equal to the value
attributeName.lt=value	Rightmost attribute is less than the value
attributeName.lte=value	Rightmost attribute is less than or equal to the value
attributeName.in=value,[value]	Rightmost attribute contains at least one value of the list
attributeName.nin=value,[value]	Rightmost attribute does not contains any value of the list

4.3.6 Continuous Optimization

Upon now, the conventional test planning of the NSs is utterly based on the developer. The continuous optimization service of the NFV Marketplace introduces dynamicity and adaptability in the operation of the V&V framework, as it comes to fulfill the integration of V&V platform in the 5G infrastructure. The NFV Marketplace targets in the assessment of the test performances while considering the actual performance of the deployed VNFs/NS in the production environment. Potential differences observed in the performance of the production and the test environment comprises a basis for extending the strain of the VNFs/NS through the proposal of new test schedules to the V&V framework.

By means of proposing new test plans, the developer is equipped with innovative tools of improving the design of the VNFs/NS. The test suggestions do not include only pre–defined script configurations, but also produce innovative tests to explore potential performance bottlenecks. For this reasons, monitoring information both from the test and the production infrastructure are collected, along with previous test configurations parameters. After the execution of the new test configurations and the gathering of the corresponding data, the mechanism will propose new V&V tests to be triggered to the V&V framework and thus causing a "feedback loop". The activity of the V&V developer is recorded, in the continuous optimization service. Based on this activity, the NFV Marketplace proposes unprecedented test scenarios, according to the user history in terms of VNF instantiations, corresponding test selections and results.

4.4 Numerical Results

In this section, a set of results evaluates the proposed approach with the aim of quantifying the performance differences of the proposed NFV Marketplace with the deployment of MongoDB in different usage scenarios. The time performance is chosen as a metric as it paramountly contributes in the latency of the real-time interactivity of the services. The presented performance results of the NFV Marketplace are measured through the time-to-live (TTL) of the implemented RESTful API. As for MongoDB, the time results are derived from direct access. Consequently, we examine the respective overhead introduced from the NFV Marketplace with respect to the plain deployment of MongoDB. On top of that, the dependency of the number of nested documents in the JSON descriptors, referred as JSON Depth, is studied as it is exponentially proportional to the amount of information. The presented results were obtained from the deployment of the NFV Marketplace in an actual 5G ecosystem [193].

The first scenario is designed to assess the performance of the CRUD operations in stressful task deployments of the NFV Marketplace, loaded with 10^3 JSON documents. Table 4.4 depicts the mean time performance of the CRUD operations of the NFV Marketplace and conventional MongoDB deployment as a function of the several JSON document depths. As expected, the time performance of the NFV Marketplace is negligibly lower than the performance of MongoDB in consideration of the added-value analytical services, delivered from the former. All operations maintain stable time performances, aside from the performance of the POST operation which is directly proportional to the JSON depth. This is particularly due to the processing of content mapping for the functionality of the full-text search and the direct manipulation of hierarchical-deep JSON documents. It has to be highlighted that all operations include the execution of processing for the update, the retrieval and the deletion of the IF in the NFV Marketplace.

In order to visualize this advantage of the performant full-text searches of the NFV Marketplace, Table 4.5 lists the mean time performance results for retrieving 10^2 JSON documents with both the existence and non-existence of indexes in the MongoDB. In case of an existing index, the obligation of predefining an utterly known field path in the JSON structure is necessary, which becomes intolerable in large documents. Also, in production environments with variable document schemas, an utterly known field path is inconceivable. In case of non-existing index, MongoDB is considered as a incapable mean of document retrieval since it is not aware of the indexed fields. However, the NFV Marketplace can efficiently execute full-text search queries in schema-agnostic documents without the necessity of indexes with stable mean time performance results.

To continue with, the performance scale and the latency of the NFV Marketplace are examined in a realistic scenario of consecutive uploads of documents. Figure 4.9 depicts the mean time performance of the POST operation as a function of the hosted documents in the NFV Marketplace for several JSON depths. The performance scale of the NFV Marketplace is direct proportional to the increase of the hosted documents and the JSON depth of documents. A further analysis of the POST operation is included in Figure 4.10, where the minimum and the maximum observation of the mean time performance is depicted for 40 level of JSON depth documents. Evidently, the range of the extreme values is approximately equal both in the NFV Marketplace and the MongoDB, rendering the former consistently reliable in its performance.

Figure 4.11 depicts the mean time performance of the PUT operation as a function of the JSON depth of the update operation in the NFV Marketplace and MongoDB. As mentioned above, the problem of non-indexed fields of MongoDB in the JSON documents results in update restrictions. On the other hand, as visualized,

Table 4.4: *Mean time performance of CRUD operations of NFV Marketplace and MongoDB loaded with* 10^3 *documents as a function of the JSON Depth*

	GET Method		**POST Method**		**PUT Method**		**DELETE Method**	
	NFV Marketplace	MongoDB	NFV Marketplace	MongoDB	NFV Marketplace	MongoDB	NFV Marketplace	MongoDB
JSON Depth	Time (ms)	Time (ms)	Time (ms)	Time (ms)	Time (ms)	Time (ms)	Time (ms)	Time (ms)
2	16.21	8.11	19.93	11.34	17.24	11.46	18.40	11.17
4	15.41	8.25	19.97	11.73	17.11	12.50	17.93	11.37
20	14.77	8.15	18.19	11.58	17.84	10.59	18.89	10.93
40	14.86	8.17	20.86	11.12	17.95	11.60	18.68	11.30
60	14.60	8.27	21.85	11.51	17.35	11.31	19.57	11.31
80	16.01	8.52	20.87	12.03	17.13	11.68	18.35	11.69

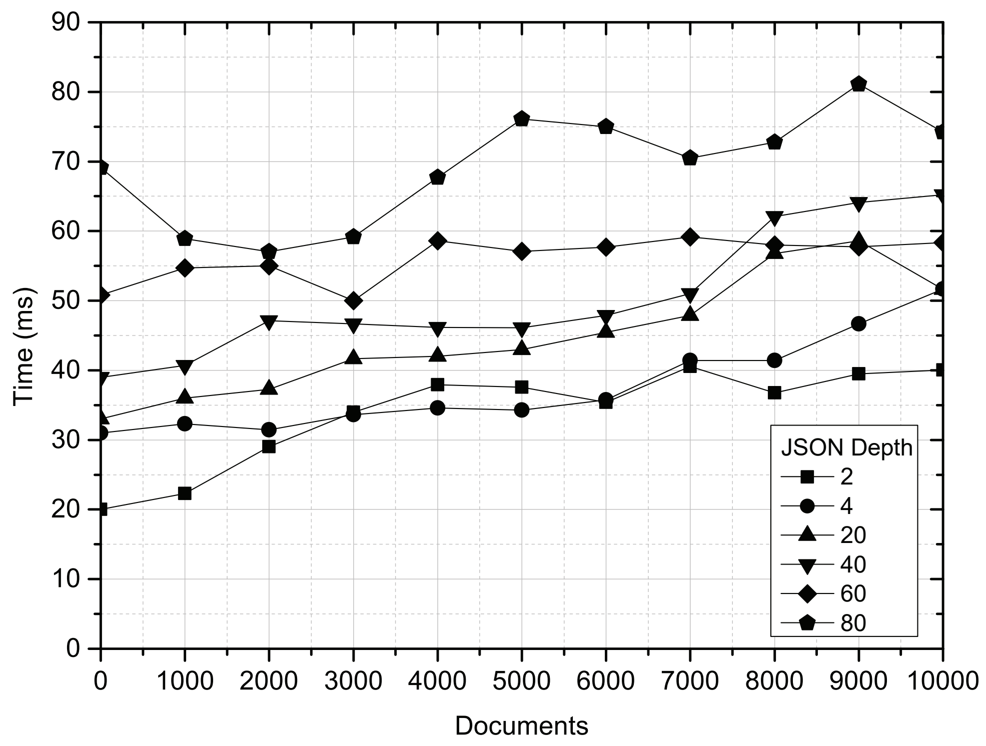


Figure 4.9: *Mean time performance of the POST operation of a single document as a function of the hosted number of documents for several depths of the JSON documents in the NFV Marketplace*

NFV Marketplace is able to update every field of the deep-hierarchic structure of the JSON documents, even in the deepest ones. Additionally, the statistical samples of the operation reveal that the variation of the time performance is approximately equal both in the NFV Marketplace and the MongoDB, even for different levels of the JSON Depth.

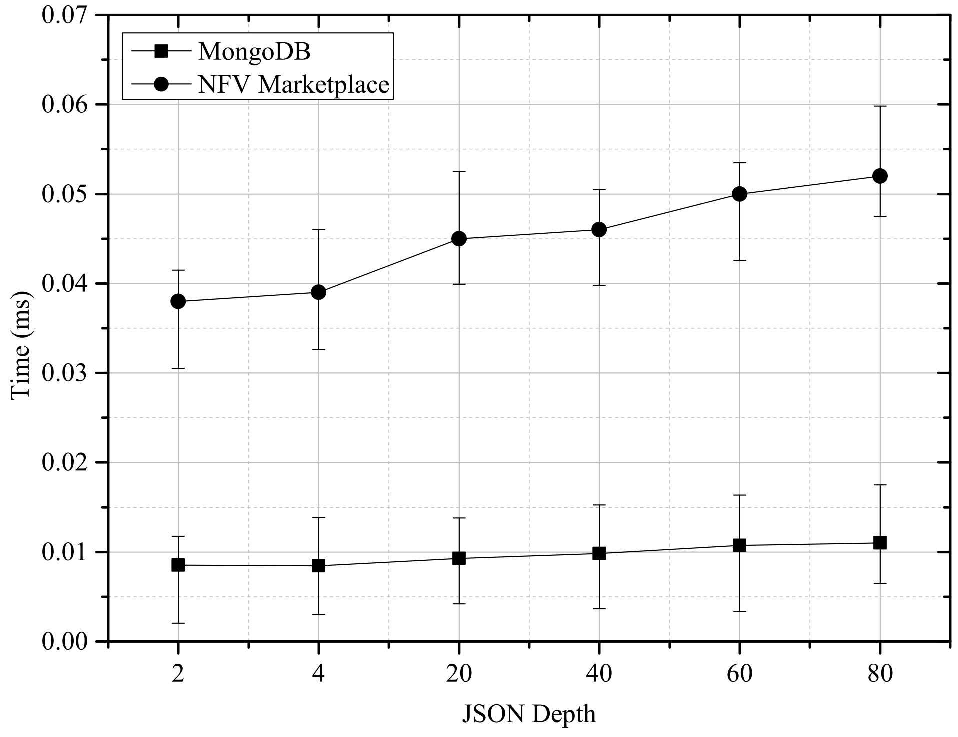


Figure 4.10: *Minimum and maximum observations of the mean time performance of the POST operation as a function of the JSON Depth in the NFV Marketplace and MongoDB*

Table 4.5: *Mean time performance of full-text search of NFV Marketplace and MongoDB through indexed and unindexed fields as a function of the JSON Depth*

	Full-text search			
	NFV Marketplace		MongoDB	
	Indexed	Unindexed	Indexed	Unindexed
JSON Depth	Time (ms)	Time (ms)	Time (ms)	Time (ms)
2	20.31	21.65	14.61	∞
4	22.54	22.21	16.65	∞
20	35.25	36.14	25.43	∞
40	42.02	41.92	29.11	∞
60	50.25	51.54	39.82	∞
80	61.78	60.55	48.75	∞

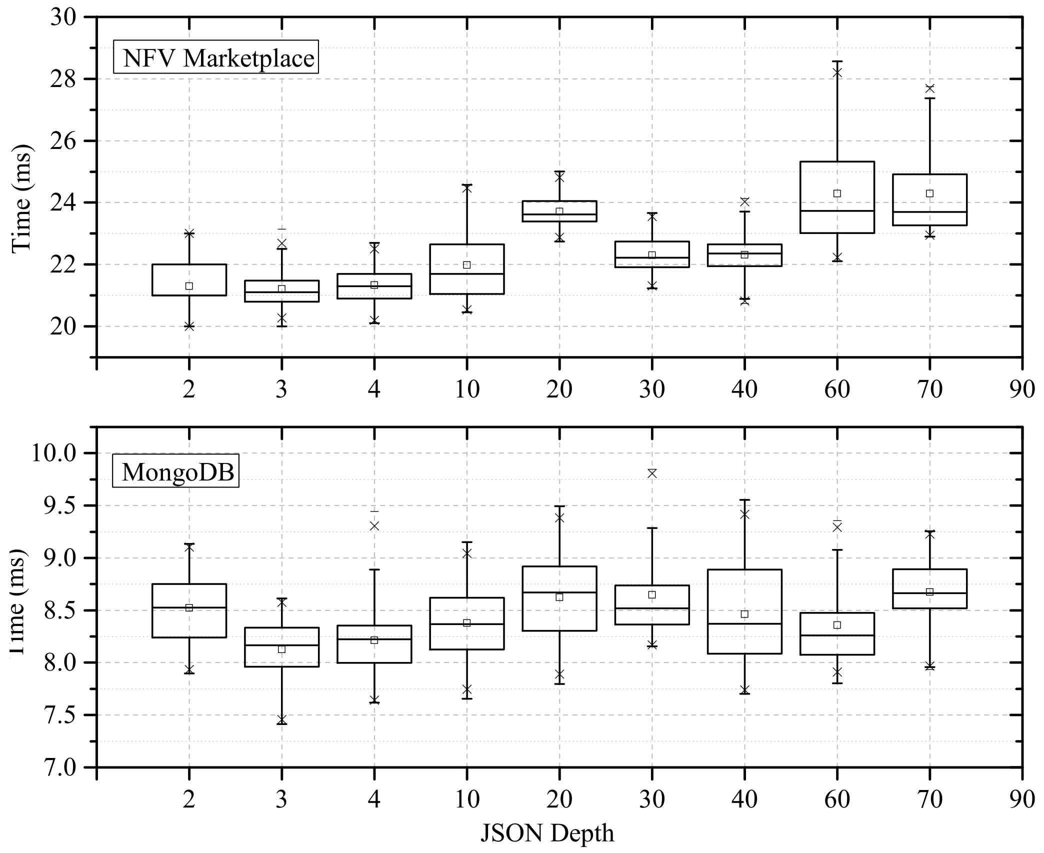


Figure 4.11: *Mean time performance of the PUT operation as a function of the JSON Depth in the NFV Marketplace and MongoDB*

Chapter 5

Conclusions and Future Work

5.1 General Overview and Conclusions

5G network architecture and operation will require a holistic and comprehensive view of the network, in order to succeed in providing high quality services, making the most out of the data analytics capabilities introduced in the complex, heterogeneous, forthcoming, 5G environments. This MSc dissertation presented the development of a novel enabling framework, the *NFV Marketplace*, allowing the diverse stakeholders the key storage component not only to deploy Network Services, but also to offer them to their customers added-value functionalities. This holistic framework supports the lifecycle management and storage of network services, as well as data analytics algorithms as novel services. The second chapter presented an introduction to the topic of 5G Networks and its relation with the principles of SDN and NFV. In more detail, it describes the proposed architectures, along with core elements of these principles. In the third chapter, a general review of the NoSQL databases was provided. More specifically, a variety of core properties from SQL databases were presented with the connection to the NoSQL ecosystem. Additionally, the NoSQL databases were meticulously studied, accompanied by their properties and concepts. In the fourth chapter, we introduced a novel prototyping NFV marketplace that goes beyond the plain NFV data repository to an intelligent trading agent of cloud services. The results of our experiments indicate an negligible overhead in the time performance of the underlying storage while delivering added-value mechanisms in the development, testing and trading lifecycles of the services. It is envisioned that the NFV Marketplace will be an integral part towards the network service development and advertising in the 5G ecosystem.

5.2 Issues for future research

This MSc thesis aimed at prototyping the development of an NFV Marketplace as an intelligent trading agent of cloud services. Directly, future issues are created that need to be researched for the further characterization of the NFV Marketplace. More detail:

- An important issue is the evaluation of the NFV Marketplace with diverse NoSQL Databases. It is substantially interesting to obtain numerical results for the performance assessment with the integration of other NoSQL Databases from MongoDB.
- As Big Data are based on the replication and distributed models, numerical results need to be extracted for the evaluation of the dependencies on these two factors.
- An integral action needs the incorporation of other machine learning and data analytics techniques to gain multiple more insights from the stored information of the NFV Marketplace.

Bibliography

[1] C. V. N. I. Cisco, "Global mobile data traffic forecast update, 2013–2018," *white paper*, 2014.

[2] M. Carugi, B. Hirschman, and A. Narita, "Introduction to the itu-t ngn focus group release 1: target environment, services, and capabilities," *IEEE Communications Magazine*, vol. 43, no. 10, pp. 42–48, 2005.

[3] J. R. Evaristo, K. C. Desouza, and K. Hollister, "Centralization momentum: the pendulum swings back again," *Communications of the ACM*, vol. 48, no. 2, pp. 66–71, 2005.

[4] H. Hu, J. Bi, T. Feng, S. Wang, P. Lin, and Y. Wang, "A survey on new architecture design of internet," in *2011 International Conference on Computational and Information Sciences*. IEEE, 2011, pp. 729–732.

[5] M. Castrucci, M. Cecchi, F. D. Priscoli, L. Fogliati, P. Garino, and V. Suraci, "Key concepts for the future internet architecture," in *Future Network & Mobile Summit (FutureNetw), 2011*. IEEE, 2011, pp. 1–10.

[6] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and J. Van Der Merwe, "The case for separating routing from routers," in *Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*. ACM, 2004, pp. 5–12.

[7] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford, "In vini veritas: realistic and controlled network experimentation," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4, pp. 3–14, 2006.

[8] S. Barkai, N. Shelef, G. Kaempfer, A. Noy, E. Bar-Eli, and R. Sidi, "Network control using software defined flow mapping and virtualized network functions," Aug. 14 2014, uS Patent App. 14/178,560.

[9] J. G. Andrews, S. Buzzi, W. Choi, S. V. Hanly, A. Lozano, A. C. Soong, and J. C. Zhang, "What will 5g be?" *IEEE Journal on selected areas in communications*, vol. 32, no. 6, pp. 1065–1082, 2014.

[10] S. Abdelwahab, B. Hamdaoui, M. Guizani, and T. Znati, "Network function virtualization in 5g," *IEEE Communications Magazine*, vol. 54, no. 4, pp. 84–91, 2016.

[11] C. Liang, F. R. Yu, and X. Zhang, "Information-centric network function virtualization over 5g mobile wireless networks," *IEEE network*, vol. 29, no. 3, pp. 68–74, 2015.

[12] M. Agiwal, A. Roy, and N. Saxena, “Next generation 5g wireless networks: A comprehensive survey,” *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 1617–1655, 2016.

[13] A. Gupta and R. K. Jha, “A survey of 5g network: Architecture and emerging technologies,” *IEEE access*, vol. 3, pp. 1206–1232, 2015.

[14] P. Mell, T. Grance *et al.*, “The nist definition of cloud computing,” 2011.

[15] S. Rowshanrad, S. Namvarasl, V. Abdi, M. Hajizadeh, and M. Keshtgary, “A survey on sdn, the future of networking,” *Journal of Advanced Computer Science & Technology*, vol. 3, no. 2, pp. 232–248, 2014.

[16] J. Zander and R. Forchheimer, “The softnet project: a retrospect,” in *Electrotechnics, 1988. Conference Proceedings on Area Communication, EUROCON 88., 8th European Conference on*. IEEE, 1988, pp. 343–345.

[17] N. Feamster, J. Rexford, and E. Zegura, “The road to sdn: an intellectual history of programmable networks,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[18] A. T. Campbell, I. Katzela, K. Miki, and J. Vicente, “Open signaling for atm, internet and mobile networks (opensig’98),” *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 1, pp. 97–108, 1999.

[19] A. Doria, F. Hellstrand, K. Sundell, and T. Worster, “General switch management protocol (gsmp) v3,” Tech. Rep., 2002.

[20] J. Biswas, A. A. Lazar, J.-F. Huard, K. Lim, S. Mahjoub, L.-F. Pau, M. Suzuki, S. Torstensson, W. Wang, and S. Weinstein, “The ieee p1520 standards initiative for programmable network interfaces,” *IEEE Communications Magazine*, vol. 36, no. 10, pp. 64–70, 1998.

[21] S. Paul, J. Pan, and R. Jain, “Architectures for the future networks and the next generation internet: A survey,” *Computer Communications*, vol. 34, no. 1, pp. 2–42, 2011.

[22] T. Lakshman, T. Nandagopal, R. Ramjee, K. Sabnani, and T. Woo, “The softrouter architecture,” in *Proc. ACM SIGCOMM Workshop on Hot Topics in Networking*, vol. 2004. Citeseer, 2004.

[23] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, “Network configuration protocol (netconf),” Tech. Rep., 2011.

[24] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, “Ethane: Taking control of the enterprise,” in *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4. ACM, 2007, pp. 1–12.

[25] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker, “Sane: A protection architecture for enterprise networks.” in *USENIX Security Symposium*, vol. 49, 2006, p. 50.

[26] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern, “Forwarding and control element separation (forces) protocol specification,” Tech. Rep., 2010.

[27] Y. Cheng, V. Ganti, and V. Lubsey, "Open data center alliance usage model: Software-defined networking rev. 2.0," *Open Data Center Alliance*, 2014.

[28] H. Song, "Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 127–132.

[29] C. Beckmann, J. Tonsing, and B. Mack-Crane, "Charter: Forwarding abstractions working group," *Rapport technique*, 2013.

[30] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[31] B. Pfaff and B. Davie, "The open vswitch database management protocol," Tech. Rep., 2013.

[32] M. Smith, M. Dvorkin, Y. Laribi, V. Pandey, P. Garg, and N. Weidenbacher, "Opflex control protocol," *IETF, Apr*, 2014.

[33] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 44–51, 2014.

[34] M. Suñé, V. Alvarez, T. Jungel, U. Toseef, and K. Pentikousis, "An openflow implementation for network processors," in *2014 Third European Workshop on Software Defined Networks (EWSDN)*. IEEE, 2014, pp. 123–124.

[35] D. Parniewicz, R. Doriguzzi Corin, L. Ogrodowczyk, M. Rashidi Fard, J. Matias, M. Gerola, V. Fuentes, U. Toseef, A. Zaalouk, B. Belter *et al.*, "Design and implementation of an openflow hardware abstraction layer," in *Proceedings of the 2014 ACM SIGCOMM workshop on Distributed cloud computing*. ACM, 2014, pp. 71–76.

[36] B. Belter, D. Parniewicz, L. Ogrodowczyk, A. Binczewski, M. Stroiñski, V. Fuentes, J. Matias, M. Huarte, and E. Jacob, "Hardware abstraction layer as an sdn-enabler for non-openflow network equipment," in *Software Defined Networks (EWSDN), 2014 Third European Workshop on*. IEEE, 2014, pp. 117–118.

[37] B. Belter, A. Binczewski, K. Dombek, A. Juszczyk, L. Ogrodowczyk, D. Parniewicz, M. Stroiñski, and I. Olszewski, "Programmable abstraction of datapath," in *Software Defined Networks (EWSDN), 2014 Third European Workshop on*. IEEE, 2014, pp. 7–12.

[38] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: Towards a model-driven sdn controller architecture," in *2014 IEEE 15th International Symposium on*. IEEE, 2014, pp. 1–6.

[39] A. Singla and B. Rijsman, "Opencontrail architecture document," *URL http://www. opencontrail. org/opencontrail-architecture-documentation*.

[40] K. Phemius, M. Bouet, and J. Leguay, "Disco: Distributed multi-domain sdn controllers," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE, 2014, pp. 1–4.

[41] D. Erickson, "The beacon openflow controller," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 13–18.

[42] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks." *Hot-ICE*, vol. 12, pp. 1–6, 2012.

[43] E. N. Maestro, "A system for scalable open flow control," *Technical Report of Rice University*, 2011.

[44] R. Wallner and R. Cannistra, "An sdn approach: quality of service using big switch's floodlight open-source controller," *Proceedings of the Asia-Pacific Advanced Network*, vol. 35, pp. 14–19, 2013.

[45] H. Yin, H. Xie, T. Tsou, D. Lopez, P. Aranda, and R. Sidi, "Sdni: A message exchange protocol for software defined networks (sdns) across multiple domains," *IETF draft, work in progress*, 2012.

[46] J. Dix, "Clarifying the role of software-defined networking northbound apis," *Network*, vol. 4, p. 11, 2013.

[47] I. Guis, "The sdn gold rush to the northbound api," 2012.

[48] B. Salisbury, "The northbound api-a big little problem," 2012.

[49] G. Ferro, "Northbound api, southbound api, east/north lan navigation in an openflow world and an sdn compass," 2012.

[50] B. Casemore, "Northbound api: The standardization debate," 2012.

[51] I. Pepelnjak, "Sdn controller northbound api is the crucial missing piece," 2012.

[52] S. Johnson, "A primer on northbound apis: Their role in a software-defined network," *SearchSDN, December*, 2012.

[53] R. G. Little, ""onf to standardize northbound api for sdn applications?" 2013.

[54] M. Yu, A. Wundsam, and M. Raju, "Nosix: A lightweight portability layer for the sdn os," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 28–35, 2014.

[55] R. Chua, "Openflow northbound api: A new olympic sport," 2012.

[56] M. S. Farooq, S. A. Khan, F. Ahmad, S. Islam, and A. Abid, "An evaluation framework and comparative analysis of the widely used first programming languages," *PloS one*, vol. 9, no. 2, p. e88941, 2014.

[57] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Practical declarative network management," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*. ACM, 2009, pp. 1–10.

[58] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM Sigplan Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[59] A. Voellmy and P. Hudak, "Nettle: Taking the sting out of programming network routers," in *International Symposium on Practical Aspects of Declarative Languages*. Springer, 2011, pp. 235–249.

[60] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *ACM SIGPLAN Notices*, vol. 47, no. 1. ACM, 2012, pp. 217–230.

[61] A. Voellmy, H. Kim, and N. Feamster, "Procera: a language for high-level reactive network control," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 43–48.

[62] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker *et al.*, "Composing software defined networks." in *NSDI*, vol. 13, 2013, pp. 1–13.

[63] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Hierarchical policies for software defined networks," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 37–42.

[64] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi, "Tierless programming and reasoning for software-defined networks." in *NSDI*, vol. 14, 2014, pp. 519–531.

[65] N. P. Katta, J. Rexford, and D. Walker, "Logic programming for software-defined networks," in *Workshop on Cross-Model Design and Validation (XLDI)*, vol. 412, 2012, p. 332.

[66] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: simplifying sdn programming using algorithmic policies," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 87–98.

[67] R. Soulé, S. Basu, R. Kleinberg, E. G. Sirer, and N. Foster, "Managing the network with merlin," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM, 2013, p. 24.

[68] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. J. Jackson *et al.*, "Network virtualization in multi-tenant datacenters." in *NSDI*, vol. 14, 2014, pp. 203–216.

[69] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," in *ACM SIGPLAN Notices*, vol. 49, no. 1. ACM, 2014, pp. 113–126.

[70] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "Fattire: Declarative fault tolerance for software-defined networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 109–114.

[71] R. Wang, D. Butnariu, J. Rexford *et al.*, "Openflow-based server load balancing gone wild." *Hot-ICE*, vol. 11, pp. 12–12, 2011.

[72] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-serve: Load-balancing web traffic using openflow," *ACM Sigcomm Demo*, vol. 4, no. 5, p. 6, 2009.

[73] D. Meyer, L. Zhang, and K. Fall, "Report from the iab workshop on routing and addressing," Tech. Rep., 2007.

[74] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "Optimizing rules placement in openflow networks: trading routing for better efficiency," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 127–132.

[75] J. Schulz-Zander, L. Suresh, N. Sarrar, A. Feldmann, T. Hühn, and R. Merz, "Programmatic orchestration of wifi networks," in *USENIX Annual Technical Conference*. USENIX Association, 2014, pp. 347–358.

[76] J. Vestin, P. Dely, A. Kassler, N. Bayer, H. Einsiedler, and C. Peylo, "Cloudmac: towards software defined wlans," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 16, no. 4, pp. 42–45, 2013.

[77] M. Bansal, J. Mehlman, S. Katti, and P. Levis, "Openradio: a programmable wireless dataplane," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 109–114.

[78] K. Wang, Y. Qi, B. Yang, Y. Xue, and J. Li, "Livesec: Towards effective security management in large-scale production networks," in *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*. IEEE, 2012, pp. 451–460.

[79] S. W. Shin, P. Porras, V. Yegneswara, M. Fong, G. Gu, and M. Tyson, "Fresco: Modular composable security services for software-defined networks," in *20th Annual Network & Distributed System Security Symposium*. NDSS, 2013.

[80] R. Hand, M. Ton, and E. Keller, "Active security," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM, 2013, p. 17.

[81] A. Arefin, V. K. Singh, G. Jiang, Y. Zhang, and C. Lumezanu, "Diagnosing data center behavior flow by flow," in *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*. IEEE, 2013, pp. 11–20.

[82] S. A. Mehdi, J. Khalid, and S. A. Khayam, "Revisiting traffic anomaly detection using software defined networking," in *International workshop on recent advances in intrusion detection*. Springer, 2011, pp. 161–180.

[83] P. Wette and H. Karl, "Which flows are hiding behind my wildcard rule?: adding packet sampling to openflow," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 541–542, 2013.

[84] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "Opentm: traffic matrix estimator for openflow networks," in *International Conference on Passive and Active Network Measurement*. Springer, 2010, pp. 201–210.

[85] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

[86] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takács, and P. Sköldström, "Scalable fault management for openflow," in *Communications (ICC), 2012 IEEE international conference on*. IEEE, 2012, pp. 6606–6610.

[87] N. Operators, "Network functions virtualization, an introduction, benefits, enablers, challenges and call for action," in *SDN and OpenFlow SDN and OpenFlow World Congress*, 2012.

[88] N. ETSI, "Etsi network functions virtualisation (nfv) industry standards (isg) group draft specifications," *http://docbox.etsi.org/ISG/NFV/Open*, 2014.

[89] K. L. Berg, T. Seymour, and R. Goel, "History of databases," *International Journal of Management & Information Systems (Online)*, vol. 17, no. 1, p. 29, 2013.

[90] A. ANSI, "X3/sparc study group on dbms, interim report," *SIGMOD FDT Bull*, vol. 7, no. 2, 1975.

[91] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[92] D. D. Chamberlin and R. F. Boyce, "Sequel: A structured english query language," in *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. ACM, 1974, pp. 249–264.

[93] D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade, "Sequel 2: A unified approach to data definition, manipulation, and control," *IBM Journal of Research and Development*, vol. 20, no. 6, pp. 560–575, 1976.

[94] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys (CSUR)*, vol. 15, no. 4, pp. 287–317, 1983.

[95] J. Gray *et al.*, "The transaction concept: Virtues and limitations," in *VLDB*, vol. 81. Citeseer, 1981, pp. 144–154.

[96] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *Acm Sigact News*, vol. 33, no. 2, pp. 51–59, 2002.

[97] E. Borowsky and E. Gafni, "Generalized flp impossibility result for t-resilient asynchronous computations," in *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. ACM, 1993, pp. 91–100.

[98] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

[99] M. Fischer, N. Lynch, and M. Paterson, "ªimpossibility of distributed consensus with one faulty process, º j," 1985.

[100] M. Herlihy and N. Shavit, “The topological structure of asynchronous computability,” *Journal of the ACM (JACM)*, vol. 46, no. 6, pp. 858–923, 1999.

[101] M. Saks and F. Zaharoglou, “Wait-free k-set agreement is impossible: The topology of public knowledge,” *SIAM Journal on Computing*, vol. 29, no. 5, pp. 1449–1483, 2000.

[102] S. Gilbert and N. A. Lynch, “Perspectives on the cap theorem,” *Computer*, vol. 45, no. 2, pp. 30–36, 2012.

[103] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one fault process.” YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE, Tech. Rep., 1982.

[104] C. Dwork, N. Lynch, and L. Stockmeyer, “Consensus in the presence of partial synchrony,” *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[105] P. Dutta and R. Guerraoui, “The inherent price of indulgence,” *Distributed Computing*, vol. 18, no. 1, pp. 85–98, 2005.

[106] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, “Communication-efficient leader election and consensus with limited link synchrony,” in *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*. ACM, 2004, pp. 328–337.

[107] L. Lamport, “The part-time parliament,” *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[108] L. Lamport and M. Fischer, “Byzantine generals and transaction commit protocols,” Technical Report 62, SRI International, Tech. Rep., 1982.

[109] S. Chaudhuri, M. Herlihy, N. A. Lynch, and M. R. Tuttle, “A tight lower bound for k-set agreement,” in *Foundations of Computer Science, 1993. Proceedings., 34th Annual Symposium on*. IEEE, 1993, pp. 206–215.

[110] E. Brewer, “Pushing the cap: Strategies for consistency and availability,” *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[111] D. E. Brown, “Method for auditing data integrity in a high availability database,” Feb. 22 2011, uS Patent 7,895,501.

[112] A. Gorelik and L. Burda, “High availability database system using live/load database copies,” Jan. 10 2002, uS Patent App. 09/782,178.

[113] M. Makpangou, G. Pierre, C. Khoury, and N. Dorta, “Replicated directory service for weakly consistent distributed caches,” in *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*. IEEE, 1999, pp. 92–100.

[114] W.-E. Chen, Y.-B. Lin, and R.-H. Liou, “A weakly consistent scheme for ims presence service,” *IEEE Transactions on Wireless Communications*, vol. 8, no. 7, 2009.

[115] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch, “Application-specific conflict resolution for weakly consistent replicated databases,” Feb. 11 1997, uS Patent 5,603,026.

[116] K. Wang, X. Zhou, K. Qiao, M. Lang, B. McClelland, and I. Raicu, “Towards scalable distributed workload manager with monitoring-based weakly consistent resource stealing,” in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2015, pp. 219–222.

[117] E. Brewer, “Cap twelve years later: How the” rules” have changed,” *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[118] L. Lamport *et al.*, “Paxos made simple,” *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[119] K. P. Birman, Q. Huang, and D. Freedman, “Overcoming the ‘d’in cap: Using isis2 to build locally responsive cloud services,” *Computer*, pp. 50–58, 2011.

[120] M. Burrows, “The chubby lock service for loosely-coupled distributed systems,” in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 335–350.

[121] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, “Megastore: Providing scalable, highly available storage for interactive services.” in *CIDR*, vol. 11, 2011, pp. 223–234.

[122] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, “Pnuts: Yahoo!’s hosted data serving platform,” *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, 2008.

[123] D. Abadi, “Problems with cap, and yahoo’s little known nosql system,” *DBMS Musings*, 2010.

[124] C. Hale, “You can’t sacrifice partition tolerance,” *Retrieved February*, vol. 20, p. 2018, 2010.

[125] C. Strozzi, “Nosql-a relational database management system,” *Lainattu*, vol. 5, p. 2014, 1998.

[126] R. Elmasri and S. Navathe, *Fundamentals of database systems*. Pearson London, 2016.

[127] J. Pokorny, “Nosql databases: a step to database scalability in web environment,” *International Journal of Web Information Systems*, vol. 9, no. 1, pp. 69–82, 2013.

[128] D. Pritchett, “Base: An acid alternative,” *Queue*, vol. 6, no. 3, pp. 48–55, 2008.

[129] A. Swaroop and A. K. Singh, “A fault tolerant token-based algorithm for group mutual exclusion in distributed systems,” *International journal of electronics, circuits and systems*, vol. 2, no. 2, pp. 194–200, 2008.

[130] A. Oussous, F.-Z. Benjelloun, A. A. Lahcen, and S. Belfkih, "Comparison and classification of nosql databases for big data," *International Journal of Database Theory and Application*, vol. 6, no. 4.2013, 2013.

[131] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1. ACM, 2012, pp. 53–64.

[132] J. Travis, "Time series data mapping into a key-value database," Jun. 3 2014, uS Patent 8,745,014.

[133] J. L. Carlson, *Redis in action*. Manning Publications Co., 2013.

[134] S. Sivasubramanian, "Amazon dynamodb: a seamlessly scalable non-relational database service," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 729–730.

[135] A. Feinberg, "Project voldemort: Reliable distributed storage," in *Proceedings of the 10th IEEE International Conference on Data Engineering*, 2011.

[136] S. Tiwari, *Professional NoSQL*. John Wiley & Sons, 2011.

[137] D. Loshin, *Big data analytics: from strategic planning to enterprise integration with tools, techniques, NoSQL, and graph*. Elsevier, 2013.

[138] K. Chodorow, *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. " O'Reilly Media, Inc.", 2013.

[139] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: The Definitive Guide: Time to Relax*. " O'Reilly Media, Inc.", 2010.

[140] "Couchbase nosql database," https://www.couchbase.com/, accessed: 2018-11-6.

[141] A. Moniruzzaman and S. A. Hossain, "Nosql database: New era of databases for big data analytics-classification, characteristics and comparison," *arXiv preprint arXiv:1307.0191*, 2013.

[142] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[143] M.-J. Hsieh, L.-Y. Ho, J.-J. Wu, and P. Liu, "Data partition optimisation for column-family nosql databases," *International Journal of Big Data Intelligence*, vol. 4, no. 4, pp. 263–275, 2017.

[144] G. Weintraub and E. Gudes, "Data integrity verification in column-oriented nosql databases," in *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 2018, pp. 165–181.

[145] G. Matei and R. C. Bank, "Column-oriented databases, an alternative for analytical environment," *Database Systems Journal*, vol. 1, no. 2, pp. 3–16, 2010.

[146] I. Robinson, J. Webber, and E. Eifrem, *Graph databases*. " O'Reilly Media, Inc.", 2013.

[147] M. Frydenberg, "The chain graph markov property," *Scandinavian Journal of Statistics*, pp. 333–353, 1990.

[148] J. Webber and I. Robinson, *A programmatic introduction to neo4j*. Addison-Wesley Professional, 2018.

[149] R. H. Güting, "Graphdb: Modeling and querying graphs in databases."

[150] "Amazon neptune - fast, reliable graph database built for the cloud," https://aws.amazon.com/neptune/, accessed: 2018-11-6.

[151] M. Arenas, C. Gutierrez, and J. Pérez, "Foundations of rdf databases," in *Reasoning Web International Summer School*. Springer, 2009, pp. 158–204.

[152] C. Gutierrez, C. A. Hurtado, A. O. Mendelzon, and J. Pérez, "Foundations of semantic web databases," *Journal of Computer and System Sciences*, vol. 77, no. 3, pp. 520–541, 2011.

[153] W. W. W. Consortium *et al.*, "Rdf 1.1 concepts and abstract syntax," 2014.

[154] R. Angles and C. Gutierrez, "Querying rdf data from a graph database perspective," in *European Semantic Web Conference*. Springer, 2005, pp. 346–360.

[155] C. Gutierrez, C. Hurtado, and A. Mendelzon, "Formal aspects of querying rdf databases," in *Proceedings of the First International Conference on Semantic Web and Databases*. CEUR-WS. org, 2003, pp. 279–293.

[156] E. A. Brewer, "Towards robust distributed systems," in *PODC*, vol. 7, 2000.

[157] G. Linden, "Make data useful," 2006.

[158] M. Mayer, "In search of a better, faster, stronger web," *Proc. Velocity*, pp. 23–33, 2009.

[159] P. A. Bernstein and N. Goodman, "Multiversion concurrency control—theory and algorithms," *ACM Transactions on Database Systems (TODS)*, vol. 8, no. 4, pp. 465–483, 1983.

[160] X. Song and J. W.-S. Liu, "Maintaining temporal consistency: Pessimistic vs. optimistic concurrency control," *IEEE transactions on knowledge and data engineering*, vol. 7, no. 5, pp. 786–796, 1995.

[161] T. S. Madhulatha, "Graph partitioning advance clustering technique," *arXiv preprint arXiv:1203.2002*, 2012.

[162] I. S. Dhillon, "Co-clustering documents and words using bipartite spectral graph partitioning," in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2001, pp. 269–274.

[163] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning," *Operations research*, vol. 37, no. 6, pp. 865–892, 1989.

[164] “Redis data store,” https://redis.io/, accessed: 2018-11-30.

[165] “Riak,” http://basho.com/products/, accessed: 2018-11-30.

[166] B. Fitzpatrick, “Distributed caching with memcached,” *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[167] G. Wang and J. Tang, “The nosql principles and basic application of cassandra model,” in *Computer Science & Service System (CSSS), 2012 International Conference on*. IEEE, 2012, pp. 1332–1335.

[168] L. George, *HBase: the definitive guide: random access to your planet-size data*. ” O’Reilly Media, Inc.”, 2011.

[169] B. Iordanov, “Hypergraphdb: a generalized graph database,” in *International conference on web-age information management*. Springer, 2010, pp. 25–36.

[170] J. Aasman, “Allegro graph: Rdf triple database,” *Cidade: Oakland Franz Incorporated*, vol. 17, 2006.

[171] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[172] P. Buneman, M. Fernandez, and D. Suciu, “Unql: a query language and algebra for semistructured data based on structural recursion,” *The VLDB Journal*, vol. 9, no. 1, pp. 76–110, 2000.

[173] K. Banker, *MongoDB in action*. Manning Publications Co., 2011.

[174] M. Habeeb, *A Developer’s Guide to Amazon SimpleDB*. Pearson Education India, 1900.

[175] J. Pérez, M. Arenas, and C. Gutierrez, “Semantics and complexity of sparql,” *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 3, p. 16, 2009.

[176] F. Holzschuher and R. Peinl, “Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j,” in *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. ACM, 2013, pp. 195–204.

[177] A. Castelltort and A. Laurent, “Fuzzy queries over nosql graph databases: perspectives for extending the cypher language,” in *International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*. Springer, 2014, pp. 384–395.

[178] J. B. Kim and A. Segev, “A web services-enabled marketplace architecture for negotiation process management,” *Decision Support Systems*, vol. 40, no. 1, pp. 71–87, 2005.

[179] T. Wolf, J. Griffioen, K. L. Calvert, R. Dutta, G. N. Rouskas, I. Baldin, and A. Nagurney, “Choicenet: toward an economy plane for the internet,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 58–65, 2014.

[180] “5gtango catalogues,” 2018, [Online; accessed 1-Febraury-2019]. [Online]. Available: https://github.com/sonata-nfv/tng-cat

[181] B. A. Miller, T. Nixon, C. Tai, and M. D. Wood, “Home networking with universal plug and play,” *IEEE Communications Magazine*, vol. 39, no. 12, pp. 104–109, 2001.

[182] D. Marples and P. Kriens, “The open services gateway initiative: An introductory overview,” *IEEE Communications magazine*, vol. 39, no. 12, pp. 110–114, 2001.

[183] D. Yu, L. Mai, S. Arianfar, R. Fonseca, O. Krieger, and D. Oran, “Towards a network marketplace in a cloud.” in *HotCloud*, 2016.

[184] R. Riggio, “The empower mobile network operating system,” in *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*. ACM, 2016, pp. 87–88.

[185] T. Soenen, S. Van Rossem, W. Tavernier, F. Vicens, D. Valocchi, P. Trakadas, P. Karkazis, G. Xilouris, P. Eardly, S. Kolometsos *et al.*, “Insights from sonata: Implementing and integrating a microservice-based nfv service platform with a devops methodology,” in *NOMS2018, the IEEE/IFIP Network Operations and Management Symposium*, 2018, pp. 1–6.

[186] R. Mijumbi, J. Serrat, J.-L. Gorricho, S. Latré, M. Charalambides, and D. Lopez, “Management and orchestration challenges in network functions virtualization,” *IEEE Communications Magazine*, vol. 54, no. 1, pp. 98–105, 2016.

[187] H. Zhang, N. Liu, X. Chu, K. Long, A.-H. Aghvami, and V. C. Leung, “Network slicing based 5g and future mobile networks: mobility, resource management, and challenges,” *IEEE Communications Magazine*, vol. 55, no. 8, pp. 138–145, 2017.

[188] I. Vaishnavi, J. Czentye, M. Gharbaoui, G. Giuliani, D. Haja, J. Harmatos, D. Jocha, J. Kim, B. Martini, J. MeMn *et al.*, “Realizing services and slices across multiple operator domains,” in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2018, pp. 1–7.

[189] C. Makaya, D. Freimuth, D. Wood, and S. Calo, “Policy-based nfv management and orchestration,” in *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. IEEE, 2015, pp. 128–134.

[190] A. Alleg, T. Ahmed, M. Mosbah, R. Riggio, and R. Boutaba, “Delay-aware vnf placement and chaining based on a flexible resource allocation approach,” in *Network and Service Management (CNSM), 2017 13th International Conference on*. IEEE, 2017, pp. 1–7.

[191] B. Khodapanah, A. Awada, I. Viering, D. Oehmann, M. Simsek, and G. P. Fettweis, “Fulfillment of service level agreements via slice-aware radio resource management in 5g networks,” in *2018 IEEE 87th Vehicular Technology Conference (VTC Spring)*. IEEE, 2018, pp. 1–6.

[192] B. R. Southam, D. C. Davidson, and D. J. Grush, “Systems and methods for testing network services,” Feb. 24 2009, uS Patent 7,496,658.

[193] P. Twamley, M. Müller, P.-B. Bök, G. K. Xilouris, C. Sakkas, M. A. Kourtis, M. Peuster, S. Schneider, P. Stavrianos, and D. Kyriazis, "5gtango: An approach for testing nfv deployments," in *2018 European Conference on Networks and Communications (EuCNC)*. IEEE, 2018, pp. 1–218.

[194] J. Quittek, P. Bauskar, T. BenMeriem, A. Bennett, M. Besson, and A. Et, "Network functions virtualisation (nfv)-management and orchestration," *ETSI NFV ISG, White Paper*, 2014.

[195] G. Breiter, F. Leymann, and T. Spatzier, "Topology and orchestration specification for cloud applications (tosca): Cloud service archive (csar)," *International Business Machines Corporation*, 2012.

[196] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "Opentosca–a runtime for tosca-based cloud applications," in *International Conference on Service-Oriented Computing*. Springer, 2013, pp. 692–695.

[197] "Onap, open networking automation platform," 2017, [Online; accessed 22-September-2018]. [Online]. Available: https://www.onap.org

[198] "Sonata nfv, agile service development and orchestration in 5g virtualized networks," 2016, [Online; accessed 29-September-2018]. [Online]. Available: http://www.sonata-nfv.eu/

[199] "5gtango project, d4.1 first open-source release of the sdk toolset," 2018, [Online; accessed 22-September-2018]. [Online]. Available: https://5gtango.eu/project-outcomes/deliverables/42-d4-1-first-open-source-release-of-the-sdk-toolset.html

[200] G. Xilouris, E. Trouva, F. Lobillo, J. Soares, J. Carapinha, M. J. McGrath, G. Gardikis, P. Paglierani, E. Pallis, L. Zuccaro *et al.*, "T-nova: A marketplace for virtualized network functions," in *2014 European Conference on Networks and Communications (EuCNC)*. IEEE, 2014, pp. 1–5.

[201] H. Karl, S. Dräxler, M. Peuster, A. Galis, M. Bredel, A. Ramos, J. Martrat, M. S. Siddiqui, S. Van Rossem, W. Tavernier *et al.*, "Devops for network function virtualisation: an architectural approach," *Transactions on Emerging Telecommunications Technologies*, vol. 27, no. 9, pp. 1206–1215, 2016.

[202] R. P. Padhy, M. R. Patra, and S. C. Satapathy, "Rdbms to nosql: reviewing some next-generation non-relational database's," *International Journal of Advanced Engineering Science and Technologies*, vol. 11, no. 1, pp. 15–30, 2011.

[203] D. S. Ruiz, S. F. Morales, and J. G. Molina, "Inferring versioned schemas from nosql databases and its applications," in *International Conference on Conceptual Modeling*. Springer, 2015, pp. 467–480.

[204] R. Hecht and S. Jablonski, "Nosql evaluation: A use case oriented survey," in *Cloud and Service Computing (CSC), 2011 International Conference on*. IEEE, 2011, pp. 336–341.

[205] Y. Li and S. Manoharan, "A performance comparison of sql and nosql databases," in *Communications, computers and signal processing (PACRIM), 2013 IEEE pacific rim conference on*. IEEE, 2013, pp. 15–19.

[206] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, “Ceph: A scalable, high-performance distributed file system,” in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 307–320.

[207] A. Beloglazov, S. F. Piraghaj, M. Alrokayan, and R. Buyya, “Deploying openstack on centos using the kvm hypervisor and glusterfs distributed file system,” *University of Melbourne*, 2012.

[208] “Lizardfs,” [Online; accessed 3-October-2018]. [Online]. Available: https://lizardfs.com/

[209] “Seaweedfs,” [Online; accessed 3-October-2018]. [Online]. Available: https://github.com/chrislusf/seaweedfs

[210] J. Heichler, “An introduction to beegfs,” 2014.

[211] E. Wong, “Dynamic rematerialization: Processing distributed queries using redundant data,” *IEEE Transactions on Software Engineering*, no. 3, pp. 228–232, 1983.

[212] L. A. West Jr and T. J. Hess, “Metadata as a knowledge management tool: supporting intelligent agent and end user access to spatial data,” *Decision Support Systems*, vol. 32, no. 3, pp. 247–264, 2002.

[213] M. Papagelis, I. Rousidis, D. Plexousakis, and E. Theoharopoulos, “Incremental collaborative filtering for highly-scalable recommendation algorithms,” in *International Symposium on Methodologies for Intelligent Systems*. Springer, 2005, pp. 553–561.

[214] “5gtango decision support mechanism,” 2018, [Online; accessed 1-Febraury-2019]. [Online]. Available: https://github.com/sonata-nfv/tng-dsm