POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

A model-based methodology to define data-intensive architectures

Tesi di Laurea Magistrale in
Engineering of Computing Systems- Ingegneria Informatica

Author: **Arianna Dragoni**

Student ID: 976071
Advisor: Prof. Alessandro Margara
Co-advisors:
Academic Year: 2022-23

Abstract

In recent years, we have witnessed remarkable advancements in the field of big data management and processing. These developments have spurred the adoption of increasingly intricate and sophisticated infrastructures designed to extract information more efficiently from vast volumes of data. As a consequence, this has significantly complicated the way distributed infrastructure architectures are conceived and managed. The thesis sets out to provide data-intensive architects with a systematic methodology for selecting an appropriate set of technologies to implement a data-intensive architecture. Data-intensive systems architectures are software structures engineered for the storage and processing of large-scale data, in line with both functional and non-functional requirements. Consequently, the specific architectural design can vary, with an emphasis on the key features necessary for effective data management and processing. This underscores the importance of identifying all potential characteristics that data-intensive architectures may exhibit and aligning them with the requirements that an architect can gather during the initial design phase. The initial part of the thesis introduces a comprehensive model intended to encompass all conceivable characteristics. This model depicts the data life-cycle within the architecture, segmented into various phases. Subsequently, for each phase, a catalog of potential features that may be required (both for storage and processing purposes) is presented. Additionally, the model outlines how these features can be supplied by different systems and the extent to which they assure coverage of the required characteristics.

The methodology to implement a use-case specific architecture initially leverages the provided model to select the necessary features and determine the requisite level of coverage, all guided by the specific requirements. It then proceeds to choose the set of technologies that enables comprehensive coverage of all features and facilitates the construction of the architecture.

Keywords: data-intensive architectures, big-data systems, methodology

Abstract in lingua italiana

Negli ultimi anni, abbiamo assistito a notevoli progressi nel campo della gestione e dell'elaborazione dei big data. Questi sviluppi hanno stimolato l'adozione di infrastrutture sempre più intricate e sofisticate progettate per estrarre informazioni in modo più efficiente da grandi volumi di dati. Di conseguenza, ciò ha notevolmente complicato il modo in cui vengono concepite e gestite le architetture di infrastrutture distribuite. La tesi si propone di fornire agli architetti di sistemi ad alta intensità di dati una metodologia sistematica per selezionare un insieme appropriato di tecnologie per implementare un'architettura ad alta intensità di dati. Le architetture dei sistemi ad alta intensità di dati sono strutture software progettate per lo stoccaggio e l'elaborazione di dati su larga scala, in conformità ai requisiti funzionali e non funzionali. Di conseguenza, il design architetturale specifico può variare, con un'enfasi sulle caratteristiche chiave necessarie per una gestione ed elaborazione efficace dei dati. Ciò sottolinea l'importanza di identificare tutte le possibili caratteristiche che le architetture ad alta intensità di dati possono presentare e di allinearle ai requisiti che un architetto può raccogliere durante la fase iniziale di progettazione. La prima parte della tesi introduce un modello completo destinato a comprendere tutte le caratteristiche concepibili. Questo modello rappresenta il ciclo di vita dei dati all'interno dell'architettura, suddiviso in diverse fasi. Successivamente, per ciascuna fase, viene presentato un catalogo di possibili caratteristiche che potrebbero essere richieste (sia per scopi di archiviazione che di elaborazione). Inoltre, il documento descrive come queste caratteristiche possano essere fornite da diversi sistemi e in che misura garantiscono la copertura delle caratteristiche richieste.

La metodologia utilizza inizialmente il modello fornito per selezionare le caratteristiche necessarie e determinare il livello di copertura richiesto, il tutto guidato dai requisiti specifici. Successivamente, procede a scegliere l'insieme di tecnologie che consente una copertura completa di tutte le caratteristiche e facilita la costruzione dell'architettura.

.

Parole chiave: architetture data-intensive, sistemi di big data, metodologia

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

Introduction **1**

1 Motivation and Contributions **5**

2 A Model for Data-Intensive Architectures **9**

2.1 Components 10

2.1.1 Component Representation 10

2.1.2 Components Interactions 12

2.2 High-Level Model Description 13

2.2.1 High-Level Model's Mappings 15

2.2.2 Multiple Data-Flows 17

2.3 Features 20

3 Computation Features **23**

3.1 Data Acquisition 25

3.2 Processing Request 25

3.3 Processing Response 26

3.4 Query Knowledge 27

3.5 Processing Actions 27

3.6 Language 28

3.7 Extension 29

3.8 Parallelism 30

4 Storage Features **33**

4.1 Persisted Data Type Feature 34
4.2 Availability 36
4.3 Atomicity 39
4.4 Isolation 41
4.5 Durability 43
4.6 Scalability 45

5 Physical Components 47
5.1 Data-Centric Systems 48
5.1.1 Processing Systems 48
5.1.2 Components Mappings 50
5.2 State-Centric Systems 54
5.2.1 Database Systems 54
5.2.2 Others State-Centric Systems 56
5.2.3 Components Mappings 57

6 Methodology Overview 69

7 Methodology: Input Requirements 73
7.1 Decision-Making Criteria 74
7.1.1 Computation Criteria 74
7.1.2 Storage Criteria 75
7.2 Infrastructure Modeling 76
7.3 Data-Flow Identification 83
7.4 Data-Flow Parametrization 85

8 Methodology: Algorithm 93
8.1 Model Instance 93
8.2 Detailed Pseudo-code of the Methodology 97
8.2.1 STEP 1: From Data-Flows to Model Mapping Intances 97
8.2.2 STEP 2: From Model Mapping Instances to Architecture Mapping 100
8.2.3 STEP 3: From Systems Data-Flows to Architecture Systems 101

9 Related Work 105

10 Conclusions and future developments 107

Bibliography 109

Introduction

We define a *data-intensive architecture* as a software architecture a company adopts to collect, analyze and manage big data to obtain actionable knowledge and serve information to internal/external users according to its requirements. We consider architectures for distributed systems, which are deployed on different nodes, that may be placed in different localities. Architectures can be regarded as an ensemble of components meticulously designed to implement data life-cycles in accordance with the specific requirements of the company. We denote a *component* as a software component that represent a unitary element performing a step in the data life-cycle within the infrastructure. These components are combined within the architecture with the purpose to delineate the data life-cycle and generate essential information to meet the company's needs. Architecture describes software infrastructures that encompass multiple systems, each with its distinct primary function. In our terminology, we define a *system* as a technological component designed to enhance data processing and/or data management. These systems encompass a wide range of technologies, including databases, queuing systems, storage systems, analytics platforms, and more, which work collaboratively within the infrastructure to optimize the processes of data processing and data management. We consider *data processing* as encompassing tasks related to the manipulation and analysis of data, whereas we define *data management* as encompassing tasks related to the persistent storage and retrieval of data. For instance, Apache Spark[15] is primarily recognized as a system dedicated to data processing, whereas MySQL[10] is primarily designed for data management, albeit it can also perform processing tasks through SQL queries. Notably, these two systems can seamlessly collaborate, with Apache Spark extracting data managed by MySQL through interaction to facilitate data processing. The combination of data processing and data management tasks within the infrastructure define the data life-cycle.

Summing up, *data-intensive architectures* refer to the structures of software infrastructures. These architectures consist of individual components that outline the various stages in the life-cycle of data that the infrastructure needs to manage. These infrastructures are constructed using systems, which are combined in a manner that enables the implementation of the data life-cycle as defined by their architectures.

Tailoring the architecture to precise requirements involves selecting various components. This includes accounting for the nature of the data being gathered and the specific methods required for its management and analysis to derive valuable insights. The extent of variability in these choices hinges on the unique needs of the company or organization.

We consider a broad spectrum of requirements, covering both non-functional and functional aspects. Functional requirements are intricately tied to the nature of queries originating from external users, while non-functional requirements aim to safeguard the integrity, consistency, scalability, and latency performance of our distributed environments. We address both transactional and processing aspects of the systems, aiming to achieve a comprehensive perspective on data management and processing.

Choosing distributed environments instead of centralized ones introduces additional complexities in meeting these requirements, which translate into new requirements[37]. An example is the consistency of data distributed across different nodes, which was not present in vertically scalable database environments where data resides on the same node. Furthermore, these requirements frequently clash with each other, leading to the existence of trade-offs that necessitate choosing between different solutions when designing a data-intensive architecture. Hence the need for more complex architectures, which require to combine different strategies to optimize trade-offs.

The contribution of this thesis is to provide a software engineering methodology to simplify the process of reconciling all the requirements initially for designing a suitable data-intensive architecture and subsequently for identifying a set of systems for implementation. The methodology depends on a provided abstract model that captures the most relevant characteristics of any data-intensive architecture. This model incorporates pertinent characteristics for complex architectures, including those primarily focused on data processing without persistence, those primarily dedicated to data management, and those that combine these functions. When software architects apply the methodology to a specific architecture, they are initially guided to choose from our model the characteristics relevant to that specific architecture, based on the requirements they have gathered from the application domain. Following this, they proceed to select a set of systems for implementation that enable the coverage of all the chosen characteristics.

To fulfill this goal, we:

1. Identify a precise list of characteristics that data processing and management at scale may present;
2. Provide an abstract model that captures the key characteristics of any architecture;

3. Provide a clear mapping of requirements to model's characteristics. Different requirements may led to the selection of different components, characteristics and component's ability in providing the characterisitic; we define the selection as *activation* of some component or feature's ability.
4. Show if and how systems can implement the model's characteristics;
5. Show how to combine the different systems to encompass all the characteristics selected in the model by combining requirements.

The methodology exploits the model and mappings above to allow architects to: precisely identify their requirements based on the provided list, define one or more architectures that contain the characteristics that satisfy the identified requirement, and select systems to realize the designed architecture in practice.

To present the methodology, we start by describing presenting an overview of the components that composes the model in Chapter 1, followed by a description of each their relative features in Chapter 3 and Chapter 4. Then, requirements are described, with their related components and features. Finally, a strategy is provided to select the set of tools to implement the architecture.

1 | Motivation and Contributions

In recent years, we have witnessed extraordinary advancements in the field of big data management and processing. These advancements have driven the adoption of increasingly sophisticated and complex infrastructures aimed at extracting information more efficiently from massive volumes of data.

Different infrastructures exhibit significant diversity in their requirements, both functional and non-functional, including but not limited to the types of data to be analyzed, processing methods, and data storage perisistence. Moreover, it is essential to consider that the same infrastructure may need to interface with a variety of different stakeholders, both data producers and consumers, each with unique and specific needs. This diversification of stakeholders introduces the complexity of processing and managing data in highly customized ways to meet the specific requirements of each group. The wide range of requirements and the heterogeneity of infrastructures result in highly diversified data-intensive architectures, each designed to address specific challenges, which predominantly revolve around defining the data life-cycles and making decisions about what data should be stored within the infrastructure and how to store it efficiently. We view the process of implementing an infrastructure as a two-step approach: the design of an architecture to depict the infrastructure, and the selection and integration of systems to realize this architectural vision. As a result, there is also a need to integrate various systems to implement data-intensive architectures more effectively. In summary, we can pinpoint two primary challenges when constructing a data-intensive infrastructure: the first is designing an architecture that encompasses all the previously described complexities, and the second is assembling a set of systems that address all the identified challenges within that architecture.

The complexity highlighted above underscores the necessity for a *reference architecture*. This terms refers to an architecture that abstracts away from a concrete infrastructure, describes a class of infrastructure, and can be used to design concrete architectures for different infrastructures within this class[41]. In other words, we consider a reference architecture as an encompassing architecture that can serve as a blueprint for all other

architectures. These other architectures can be derived from specifications within the reference architecture.

Many reference architectures have been presented in the literature. By following different research approaches, [46] and [41] have both introduced reference architectures that unify the various data life-cycles that can be implemented within an infrastructure. They both conceptualize the life-cycle as a sequence of phases, starting with data acquisition and continuing with data processing to generate results. However, their reference architectures primarily delve into functional requirements, thus focusing mainly on extracting and showcasing data processing characteristics of different tasks. The data management aspect is confined to determining whether data is stored in persistent or temporary storage, and it primarily concentrates on non-functional requirements directly related to data processing. Aspects such as transactional requirements are not thoroughly examined in these works. [42] introduce a model for data-intensive systems that consolidates crucial design and implementation decisions into a cohesive framework. They take into account both data processing and data management systems in data-intensive scenarios. However, they do not discuss architectures for integrating these systems to define a complete infrastructure. In the reference work [27], numerous reference architectures are presented for various architectural paradigms, including Lambda, Kappa, and others. The analysis assesses how each of these architectures addresses a diverse range of requirements, both functional and non-functional. However, it is noteworthy that there is no single comprehensive reference architecture that encompasses all the necessary components to represent any of the mentioned reference architectures.

As a primary contribution of this thesis, we have developed a reference architecture, which we term as the *model*, that aims to encompass all the characteristics that a data-intensive architecture may possess when handling both data processing and data management tasks at each stage of the data life cycle. In fact, we will demonstrate the versatility of our model, showcasing its adaptability for architectures with a primary emphasis on data storage with transactional properties, as well as for architectures primarily geared towards data processing. Through the selection of different components, our model can be tailored to suit a variety of architectural paradigms widely documented in the literature, including Lambda, Kappa, Liquid, and others, as well as infrastructures such as OLAP (Online Analytical Processing), OLTP (Online Transaction Processing), HTAP (Hybrid Transactional/Analytical Processing), and Data Warehouses.

Furthermore, reference architectures predominantly emphasize theoretical aspects, and the existing literature lacks a systematic software engineering methodology[27] to define a data-intensive infrastructure starting from high-level requirements. In practical terms,

when faced with a specific application scenario, there is a notable absence of a formalized process for identifying an appropriate architecture and the corresponding set of technologies required for its successful implementation.

Such a formalized process could prove immensely advantageous in the domain of Infrastructure-as-Code (IaC). IaC represents a DevOps approach that revolves around the management and provisioning of infrastructure through machine-readable definition files, eliminating the need for manual physical hardware configuration or interactive configuration tools[31]. Orchestrators assume a pivotal role in this context by providing automatic support to practitioners during the development of infrastructural code. Therefore, armed with an algorithm capable of extracting the necessary systems based on a list of requirements, and with the aid of automation and the principles of Infrastructure as Code (IaC), it becomes plausible to swiftly and consistently implement the architecture, with requirements as the sole specification. In fact, IaC allows for clear definition of the necessary resources and configurations, ensuring that the infrastructure aligns with the selected technologies. By using an orchestrator to orchestrate the application, it could be possible to efficiently coordinate the various components of the application with the underlying infrastructure, ensuring everything operates harmoniously.

In this work, as our second primary contribution, we aim to establish a systematic methodology for crafting a bespoke data-intensive architecture tailored to specific application requirements. This methodology aim at guiding architects in transforming identified requirements into a tangible physical infrastructure. It can be considered an algorithmic approach that architects can readily follow to actualize their application-specific architectures. By providing the identified requirements as input, this process yields a comprehensive set of technologies necessary for the architecture's successful implementation.

2 | A Model for Data-Intensive Architectures

This chapter describes our model of a data-intensive architecture in terms of the components it encompass. Recalling our definition, our *model* serves a reference architecture that aims to encompass all the characteristics that a data-intensive architecture may possess when handling data processing and/or data management tasks within a data life-cycle. It is delineated based on the data life-cycle that can be executed within such an architecture, and it emphasizes the potential capabilities that can be offered. As a reminder, our definition characterizes a *component* as a discrete element responsible for executing a particular phase within a data life-cycle. Therefore, our model is structured as a collection of these components connected collectively outline the entire data life-cycle.

In this chapter, we start by describing the single components and their interactions in section 2.1. Then, we provide in Section 2.2 a high-level model description that combines black-box components. Finally, we describe how characteristics inside each components are represented in Section 2.3. In this chapter, we refrain from listing the characteristics associated with each component. These characteristics will be detailed in Chapter 3 and Chapter 4.

Model's Terminology We define *query* any request that has to be handled within the architecture. We distinguish between *static* and *dynamic* queries. *Static* queries are those established during the architecture implementation time, while *dynamic* queries are those generated by users during the system's usage. Dynamic queries can require both data management and data processing tasks, while static queries are primarily focused on data processing tasks only.

The model's external actors are *producers* and *consumers*. We define *producers* as external sources from which data is collected, while *consumers* as external actors that can receive static query results and/or generate dynamic queries and receive results.

The data that passes through the model and is optionally persisted can be *raw*, *merged*

or *derived*. We refer to the data received from sources as *raw*, indicating that it has not undergone any form of modification. In certain scenarios, data coming from external sources may not be compatible with each other to be processed together by the architecture's operations. In such cases, preliminary data elaborations may be required before the actual data processing and/or management tasks can commence. These elaborations ensure that the incoming data is suitably unified for subsequent processing within the architecture. We refer to the raw data that undergoes these elaborations as *merged*. Both raw and merged data contain identical information, as these operations do not generate additional data, but rather transform the raw data into a different representation that still contains the same information. Both raw and merged data can undergo processing and or management that extracts new data containing additional information. In such cases, we refer to the resulting data as *derived* data. An example of raw information could be a data-set of thermal measurements received from sensors (producers). Merged information could be the thermal measurements coming from different sensors in a unified format, and derived information, would be the computed average of these measurements.

2.1. Components

In this section, we start by offering an explanation of how each component is depicted within the model in Subsection 2.1.1. Subsequently, we delve into a description of how these components interact with one another and external actors in Subsection 2.1.1.

2.1.1. Component Representation

Each component within the model comprises two essential elements: a *phase* and a *storage*. The *phase* corresponds to the step in the big data life-cycle that the component is associated with, while the *storage* indicates whether the component needs to preserve data produced at that specific phase. Both phase and storage within each component can have one or more features. We define a *feature* as a capability that the component offer to perform data management or data processing task. Features related to a phase are defined as *computation features*, whereas those related to the storage are defined as *storage features*. The representation of the component is depicted in Figure 2.1.

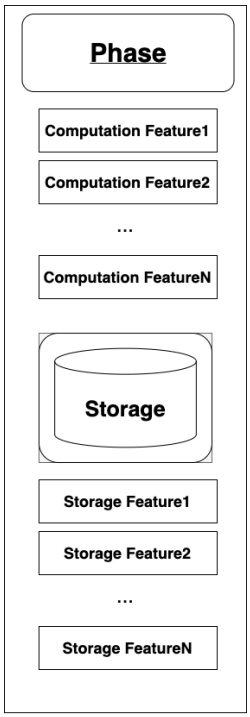


Figure 2.1: Component Overview.

Computation features are primarily associated with capabilities related to data processing tasks, while storage features are mainly associated with capabilities related to data management tasks.

State-Centric vs Data-Centric Components

We have stated that our model is structured as a collecton of components that communicates to collectively outline the entire data life-cycle, and each individual component represents a step within this life-cycle. It's important to note that it's not always necessary to retain the data produced at the end of every step in the data life-cycle; in some cases, simply transferring the data to the next step without storage is sufficient. Therefore, not all individual components necessarily need to have a storage. We say that the storage is *active* when it's necessary to preserve the data produced at the end of its data life-cycle step. In such cases, the component has storage features, indicating that it has capabilities related to data management. Conversely, when data storage is not required at the end of the step, the component neither has storage nor storage features, and can simply route the data to the next step in the life-cycle. We define *state-centric components* as those components with active storage, and *data-centric components* as those components without active storage. *State-centric* components are storage systems, mostly dedicated to persist information; *data-centric* components are processing components used to process

the data without having to persist it.

2.1.2. Components Interactions

The components within the model communicate with each other through the exchange of data and queries (both static and dynamic). Figure 2.2 shows the interaction among phases, which is characterized by two different types of flows: the *Data-Flow* entering the model from the left and moving to the right and the *Dynamic-Query-Flow* entering the model from the right and moving to the left. We define *Data-Flow* the process of transferring data from one component to another, including all computations or processing operations that occur within each component, while we define *Dynamic-Query-Flow* as the process of transferring a dynamic query from one component to another. Static queries are integrated into the data flow because they are defined within the components during the architecture's implementation. During the architecture's execution, the process primarily involves data exchange between the components themselves, as they have already implemented the necessary functions to perform the required processing and generate the data to be passed on.

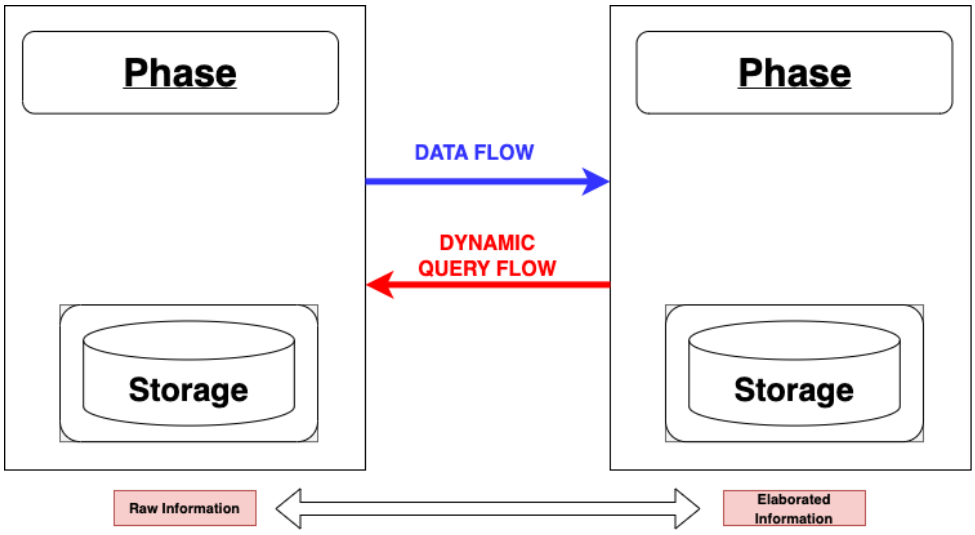


Figure 2.2: Components Interaction

In the following, Subsection 2.1.2 and Subsection 2.1.2 provide descriptions of the Data-Flow and the Dynamic-Query-Flow, respectively.

Data-Flow

Data that comes from producers is collected, and optionally persisted in the storage, by the leftmost component. Notice that for simplicity we do not distinguish whether the data is ingested from producers with some command (e.g., `INSERT`) or with a stream of data. This data, according to the information definitions provided above, is considered as *raw*. The raw data in the leftmost phase may be subject to some data elaboration (taking the form of *merged*), and optionally persisted. Starting from either raw data or, if present, merged data, management and processing tasks can be performed to obtain *derived* information and optionally persist it. These tasks can be triggered by both static and dynamic queries. As a consequence, the model manages more raw information moving on the left components, and more elaborated information on the right components: *raw* and *merged* information will be in the leftmost components, and *derived* one, when produced, in the rightmost components.

Data may flow from one component to the other either in *push* mode or in *pull* mode: with *push* mode, the data is sent to the next component with a time that can be configured by the designer; with *pull* mode, the receiver component requires available data when necessary. The choice is done when selecting the systems during the methodology, as it is the system itself that can have a push or pull implementation based on its design. In case of push mode, producer and consumer must be decoupled, so as to avoid situations in which the consumer cannot manage the speed with which the producer sends data. For this reason, push systems implements some application-level flow control mechanism to protect the system against being overloaded when data arrives too fast. For example, Flink implements a backpressure mechanism.

Dynamic-Query-Flow

Dynamic queries are ingested by consumers in the model from the rightmost phase, and flow on left phases to take the response. Answers to the queries, which are included in the data-flow, will be taken in a phase according to the asked type of information: the more raw is the requested data, the further the query will flow to the left to get the response.

2.2. High-Level Model Description

In this section, we delineate the diverse roles that the components can assume within the architecture. These roles correspond to the various steps that can be executed within a data life-cycle.

Let us first consider components as black boxes with their optional storages, without going into detail describing their capabilities (i.e., their *features*). The next chapters will then describe the features each component can have.

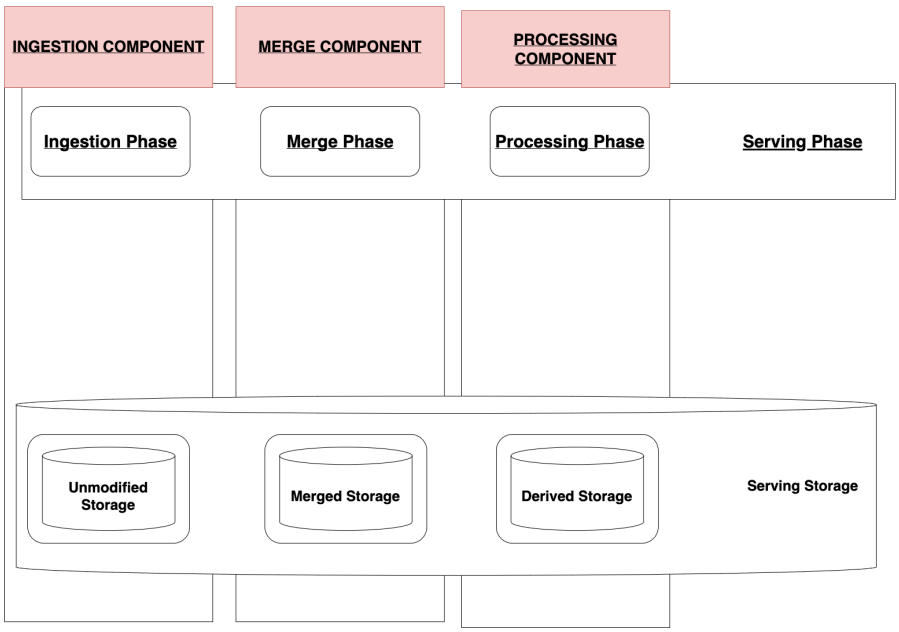


Figure 2.3: High-Level Model

Figure 2.3 illustrates the model, omitting interactions, which are depicted in Figure 2.2. Our model is composed of three main components with their phases and optional storages.

- The *Ingestion* component acquires data from external sources as it comes; it is composed of adapters that communicate with the outside world and convert the protocols used by the outside with what is in the system. The *Unaltered Storage* is used to store *raw* data as it comes from the outside, which doesn't necessarily have a defined format or schema, and it can store data in different formats.
- The *Merge* component acquires data from the previous phase. It is present if and only if some union over the *raw* data coming to different sources is necessary to create *merged* data and facilitate the work that will be done by the following phases. The *Merged Storage* can persist *merged* data produced in this phase.
- The *Processing* component is responsible for doing processing actions over the *raw*/*merged* data in order to perform more in depth analysis and produce *derived* data. For example, the average of some values that can be stored either in the unaltered storage or in the merged storage is computed in this phase, and the average

is *derived* data. The *Derived Storage* stores the *derived* data that is produced in this phase.

The *Serving* component extends across the other components to indicate their capability to perform in-memory computation (for the phases) and in-memory storage and management (for the storage).

Summing up, *raw* data can be stored in the *Unaltered Storage*. When data needs to be unified before storage, it becomes *merged* and can be stored in the *Merged Storage*. The *Derived Storage* saves *derived* data.

A specific architecture can be mapped to this model by *activating* some components and optionally their storage. In the case of data-centric component, the storage is not activated, and data flows to the next phase without being 'remembered'. Consequently, it will no longer be available in the future. For example, if the *ingestion* component is a data-centric component, the result of the query is directly sent either to the *merge* phase or to the processing phase without being persisted. In the case of state-centric component, the storage persist data produced in that phase for future usage. For example, if the processing component is state-centric, the result of the query is persisted in the derived storage before being delivered to the consumer. The serving phase can be activated only for active components, and this signifies an intention to prioritize in-memory processing whenever feasible. The serving storage can be activated only for those active components whose storage is active. In this case, the data generated by that component is stored in-memory, and the management actions are performed in-memory. However, it's important to note that this doesn't necessarily exclude the possibility of maintaining a secondary copy on disk, often for fault tolerance purposes, ensuring data durability even in the event of failures or system restarts.

Components do not necessarily represent different components of the real system. For example, a single system can first transform raw data in *merged* data, and then can process *merged* data to obtain derived data. In the model, this is split in two phases: the merge phase and the processing phase.

2.2.1. High-Level Model's Mappings

In the following, we will illustrate how to use the model described above, combining the various components to tailor it to specific architectures. In these examples, we assume that the in-memory options are always disabled. However, they can be enabled in each case to accommodate certain in-memory data lifecycle tasks. Notice that when activating a component, its phase is automatically activated.

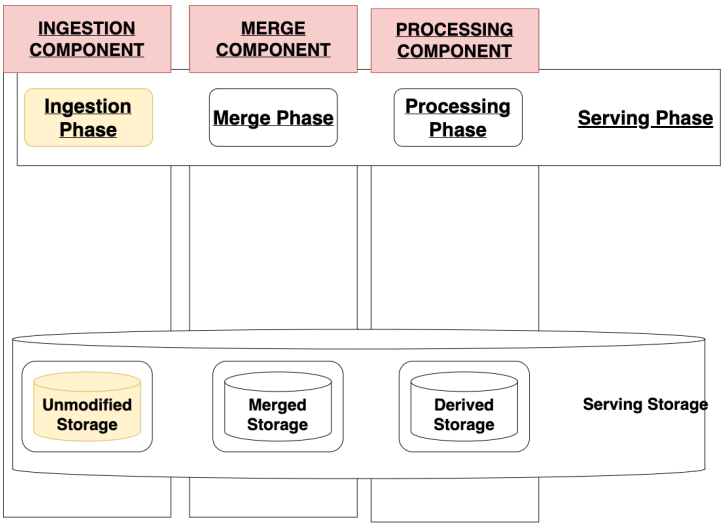


Figure 2.4: Example of Model's Mapping

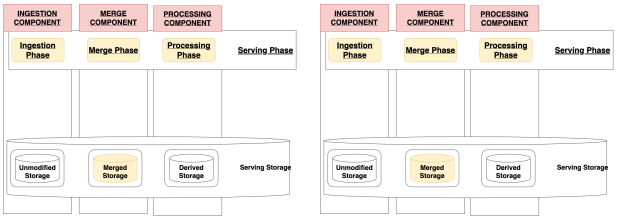
An application of the model could be an infrastructure that gather raw data from producers, with receivers making requests for this raw data in form of dynamic-on-demand queries. The architecture of the infrastructure is mapped in the model by only activating the Ingestion component with both its phase and storage, as illustrated in Figure 2.4.

Figure 2.5: Examples of Model's Mappings.

The merge component is activated when raw data ingested by different producers necessitate some form of standardization or harmonization. In this scenario, the architect has the option to retain both the unaltered storage to store raw data and the merged storage

to store merged data or choose to keep only one of them based on the collected requirements. As an example, if queries only ask for merged data, the architect may decide to keep only the merged storage (see Figure 2.5a). However, for fault tolerant reasons he/she may decide to keep also the raw version of the data stored, so that in case of crash of the physical component that implements the merged storage, it is possible to recompute the merged information starting from the raw one that is persisted in the unaltered storage. This last case is shown in Figure 2.5b. Furthermore, it might not be necessary to save the elaboration result, which corresponds to the derived data. In such cases, the architect can choose to persist only the raw data (Figure 2.5c) in the unaltered storage or to not persist anything at all (Figure 2.5d).

Moving forward, we will delve into and expand upon scenario 2.5a (i.e., only the merged data is persisted); however, the concepts discussed can be applied to any of the earlier scenarios.



(a) (b)

Figure 2.6: Examples of Model's Mappings.

Any time the query requires derived data that must be extracted through processing actions over the raw/merged data the processing component is activated. Whether to activate the derived storage of not depends on the need to persist the derived information that has been obtained. This derived information can serve as either to answer to the query or as intermediate computations required to obtain the final result. Figure 2.6a illustrates a scenario in which the derived data, including both intermediate and final results, do not require persistence. Figure 2.6b describes the situation in which the final result needs to be persisted. In this case, the derived storage is activated.

2.2.2. Multiple Data-Flows

Data-intensive architectures pertain to a data life-cycle that, due to the stakeholder diversification introduced in Chapter 1, may require different data handling approaches within the same architecture. As a result, data within the life-cycle could follow distinct paths, which we refer to as*data-flows*. In the previous chapter, we presented our model,

focusing on a life-cycle consisting of a single data flow. However, to apply our model to a real-world architecture, it might be necessary to define multiple query flows, each with its customized activations. To expand on this concept in our model, and accommodate a life-cycle with multiple data flows, we need to provisionally instantiate a separate instance of the model for each data-flow and eventually combine them to obtain the final complete architecture.

Notice that producers and consumers can be shared by different data-flows. For example, consider an online sales system: Producers and consumers are managers and users. As producers, users produce data about their personal information, what they see as they browse the site and what they buy, while managers produce data about what is for sale. As consumers, users want to see data about their purchases, what's available, and what they're recommended based on previous actions; managers want to see statistics on purchases, trends, and more. Based on this, the architect identifies two data-flows: one that produces output for users, and one for managers. Thus, a first data-flow will have managers and users as producers and users as consumers, and a second data-flow will have managers and users as producers and managers as consumers. In this case, the different data-flows share ingestion phase and/or serving phase to obtain the final architecture.

An example is the Lambda Architecture, which is a architectural model designed to address two fundamental needs in data analysis: the need to handle complex queries that require extensive processing and the need to provide rapid responses to high-throughput queries. These two requirements necessitate distinct strategies for query management, as extensive computations take time to complete, while real-time queries demand immediate responses. Consequently, using the same systems for both types of queries would not meet these requirements. The Lambda Architecture, therefore, divides data elaborations into two separate streams: a batch-processing stream for complex queries and a real-time processing stream to handle high-throughput queries. To achieve this, it is divided into three layers [27]:

- The batch layer stores input data as batch views, which are written in a master data-set and periodically merged with new data. These batch views contain the results of complex queries.
- The speed layer processes real-time views as soon as data arrives, and send them to a storage in the serving layer. These real-time views contain the results of real-time queries.
- The serving layer collects both batch views from the batch layer and real time-views from the speed layer and produces merging layer to provide a unifying access point

of the results.

We can model the Lambda architecture by instantiating separate instances of our model for each type of query, corresponding to different data-flows, one data-flow associated to the batch-processing part, and one data flow associated to the stream-processing part.

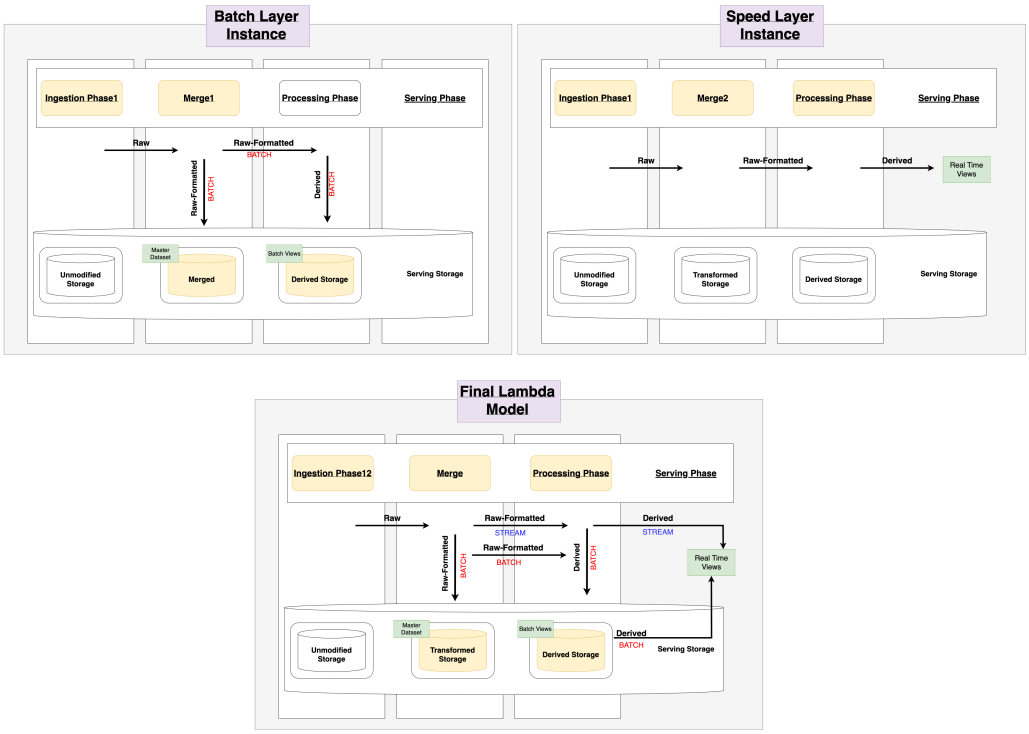


Figure 2.7: Lambda's Pipelines and Final Model

For this example, we consider that requirements are such that

- raw data comes from different producers, and need to be unified, and becomes merged data. If this is not the case, the *merge* Phase is omitted, and instead of the *merged storage* we select the *unaltered storage*;
- only merged data needs to be persisted, while raw data can be discarded. If this is not the case, the *unaltered storage* needs to be selected too.

The Lambda architecture is mapped in our model as depicted in Figure 2.7. It is composed of two instances of the model that merges together at query time.

The first flow models the batch layer. The ingestion components retrieve data from producers and sends them to the merge component. Here data is elaborated in merged data and stored in the merged storage, which corresponds to the master data-set. The processing phase periodically retrieve data from the merged storage in batch, performs computations to produce derived data and stores them in the derived storage. The second flow models the stream layer. The ingestion component retrieves data from producers and sends them to the merge component, directly produces real time views without persisting merged information. These two instances are merged to obtain the final model.

2.3. Features

Up until now, we've considered each component as a black box. In the following, we will delve deeper to examine the characteristics of each individual component. We have defined a *feature* as a capability that a component offers to perform data management or data processing tasks. Each component is composed of a collection of features, meaning that it can provide a set of capabilities to perform its tasks. The overall model, which is a collection of components composed of features, represents the complete set of potential capabilities that each data-intensive architecture can offer for implementing a data life-cycle. We have categorized features into two distinct types: Computation features and Storage features. Computation features primarily encompass capabilities related to data processing tasks, while Storage features predominantly involve capabilities related to data management tasks.

To formally define these categories, *Computation* features refer to capabilities that enable data processing within the architecture. Conversely, *Storage* features pertain to how data is stored and managed, considering specific assumptions. Computation features can be delivered by both data-centric and state-centric components, storage features be delivered only by state-centric components. We say that components can have different *approaches* in providing each feature, indicating that the component can offer that particular capability through different options.

A specific data-intensive architecture can be mapped to the model by activating, for each data-flow, its specific components, and for each of these components activate how each feature should be provided, i.e., which approaches each component should have. Activated approaches are not the only provided by the overall architecture, but they are the ones identified as essential. This means that systems implementing the infrastructure described by an architecture must necessarily include these active approaches. This doesn't directly imply that all other approaches won't be provided by the infrastructure;

it simply means that when selecting systems, we must ensure that essential approaches for providing capabilities are present.

In our model, a feature is represented as a rectangle within the component, and within it, there are rectangles that represent the activated approaches (Figure 3) .

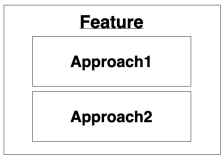


Figure 2.8: Feature Representations.

Chapter 3 and Chapter 4, provide a description of, respectively, computation and storage features, with their respective approaches. Chapter 5 describes how to extract a set of physical components by considering the activated logical components and features.

3 | Computation Features

This chapter describes the processing capabilities of model's components.

We can divide our model into two main parts: the upper part, which encompasses the computation features, and the lower part, which comprises the storage features. In this chapter, our focus is on the upper part, which contains the computation features and is illustrated in the Figure 3.1. The lower part, located beneath the storage and displaying the storage features, will be addressed in the next chapter (Chapter 4).

Computation features are depicted in green, while their corresponding abilities are represented in blue. As illustrated, components may offer varying features based on their respective phase's tasks, and each feature can be delivered through different abilities. All components with the same phase's type possess a set of features, and they differentiate themselves from one another in their specific architecture based on the abilities through which they provide each feature.

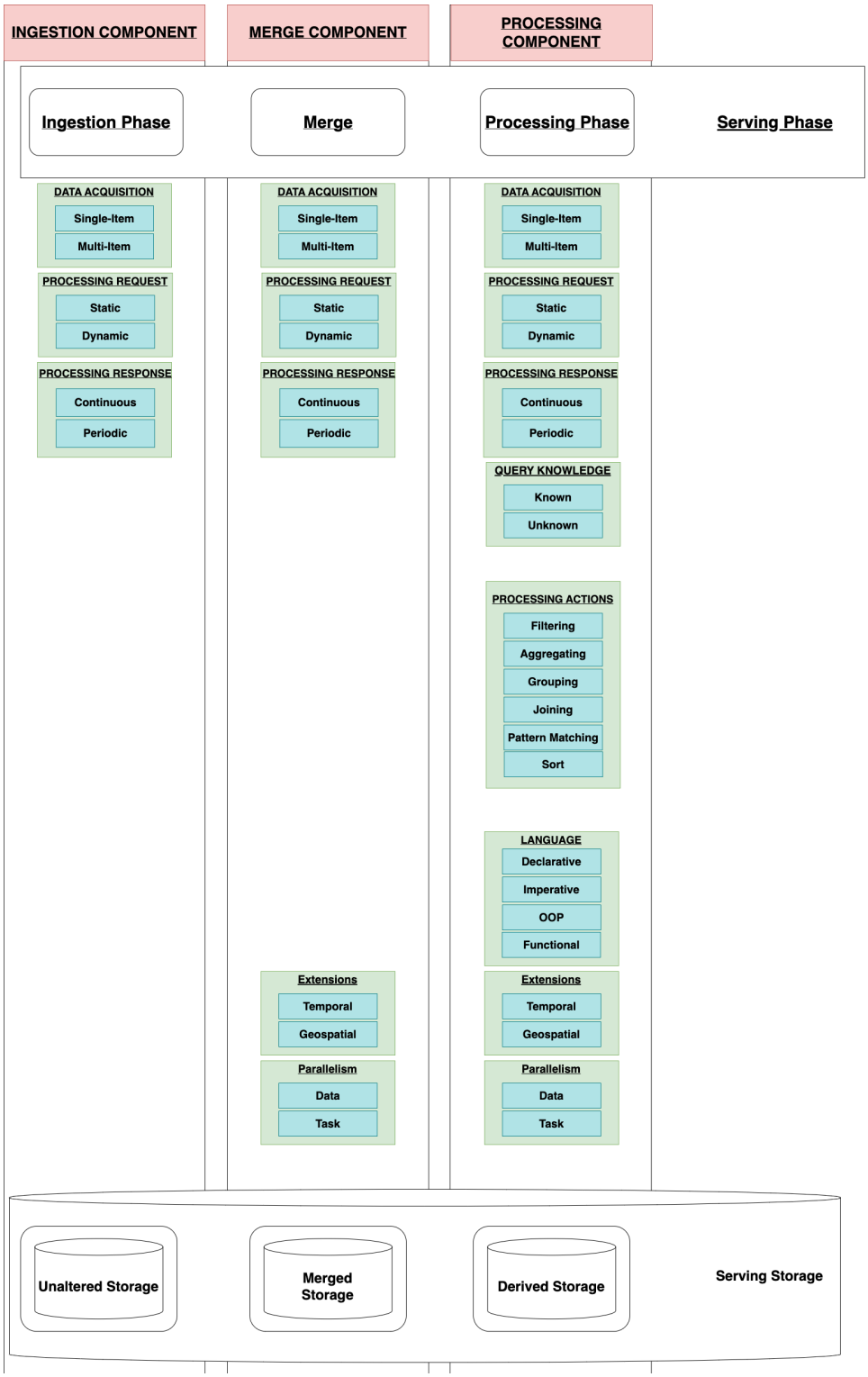


Figure 3.1: Computation Features Overview.

In the following sections, an explanation of each feature is provided, along with their respective abilities.

3.1. Data Acquisition

We define *Data Acquisition* as the component's capability to acquire data either from other components or from producers. When the component receives the data, it can either acquire it immediately upon arrival or pre-accumulate it before retrieval and processing begins. Viewing the data-set ingested in the component as a collection of items, the first approach involves acquiring one item at a time, whereas the second approach involves gathering a group of items before acquisition. As a result, we define the first approach as *single-item* and the second one as *multi-item*.

The feature representation is depicted in Figure 3.2).

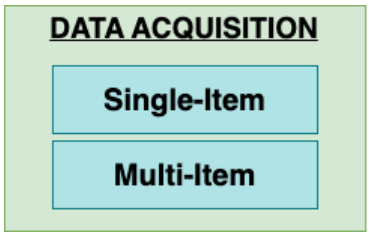


Figure 3.2: Data Acquisition Feature.

Based on the previously provided definitions, we differentiate between*Single-Item Data Acquisition* and *Multi-Item Data Acquisition* abilities in retrieving data.

3.2. Processing Request

We define *Processing Request Feature* as the capability of the component to receive a query that enters the architecture. In Chapter 2 we have characterized queries in *static* and *dynamic*. We have defined *static* queries as those established during the architecture implementation, primarily focused on data processing tasks. Conversely, we have defined *dynamic* queries as queries generated by users during the system's usage that may encompass both data management and data processing task. Hence, the first enters the architecture when it is created, while the second enters the architecture at run-time.

The feature's representation in our model is shown in Figure 3.3.

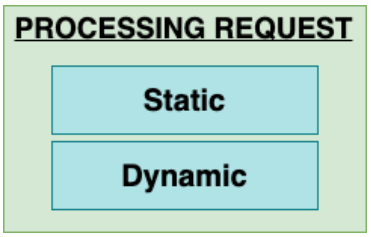


Figure 3.3: Processing Request Feature.

We define *Static Processing Request* as the component's approach to handling static requests, while *Dynamic Processing Request* as the component's approach to handling dynamic requests. We can state that *static* queries are triggered by the component as soon as the architecture is created, while *dynamic* queries enter the component when they are triggered either by a consumer or by another component within the model.

3.3. Processing Response

We define the *Processing Response Feature* as the capability of the component to send out data requested either by the consumer of by another component. We differentiate between two primary modes for processing responses to be sent: the ability to respond periodically to requests and the ability to provide continuous responses. In the first case, the component sends a response at predefined time intervals, whereas in the second case, it delivers a continuous stream of data in response

Accordingly, we categorize processing response abilities into *Periodic* and *Continuous* abilities, as it is depicted in Figure 3.4.

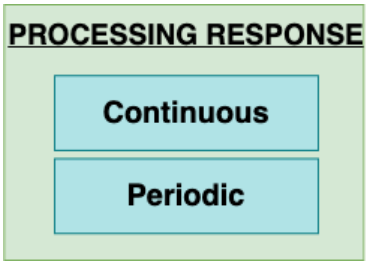


Figure 3.4: Processing Response Feature.

We define *Periodic Processing Response* as the component's approach to transmitting periodic answers, while *Continuous Processing Response* as the component's approach to proactively sending out data as soon as new data becomes available. This external interaction can involve either a consumer or another component within the system.

3.4. Query Knowledge

We define the *Query Knowledge Feature* as the component's capability to manage queries that may be known at various points in time. We differentiate between two primary architectural abilities: the architecture may be capable of handling both known queries at the time of implementation, which are planned and configured in advance, and unknown queries, which can emerge unexpectedly. Known queries can be handled by all types of components because they are specified in advance, and components are designed and configured based on these queries. On the other hand, flexibility in handling unknown queries can be an important aspect to consider during the design of data processing systems because architectures can be designed with varying degrees of flexibility in managing unknown queries. In general, we can say that all state-centric components can respond to unknown queries as soon as the necessary data to obtain the answer is present in storage, while data-centric components are less predisposed to do so because data is not persisted.

Figure 3.5 illustrates the feature, which can be present in any component of the model.

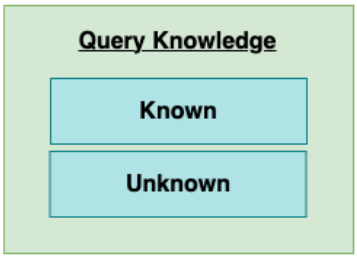


Figure 3.5: Query Behavior Feature.

In our model, we distinguish between *Known Query Behavior* and *Unknown Query Behavior*. We define *Known Query Behaviour* as the ability to handle queries that are known at the time of implementing the component, and *Unknown Query Behavior* as the ability to handle queries that may emerge in the future.

3.5. Processing Actions

We define *Processing Actions Feature* as the capability of the component to provide actions to process the data and extract derived information. This feature is present only in components whose phase is the Processing Phase, as it includes the ability to perform various data manipulation operations, each with a specific goal, which can be combined to obtain derived information.

Figure 3.6 depicts the feature's representation in our model, where we distinguish between *Filtering*, *Aggregation*, *Grouping*, *Joining*, *Sorting*, *Pattern Matching* abilities.

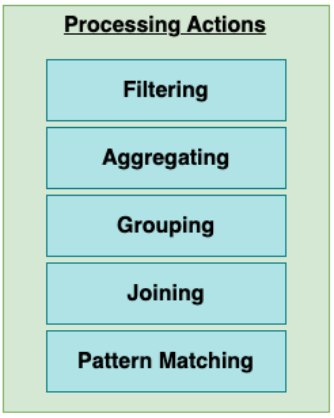


Figure 3.6: Processing Actions Feature.

We define *Filtering* as the ability to select relevant or of interest data, eliminating those that are unnecessary or do not conform to defined criteria. We define *Aggregating* as the ability to combine data from different sources or categories, allowing for the calculation of statistics or simplification of data comprehension. We define *Grouping* as the ability to facilitate the organization of data based on common attributes, enabling analysis and visualization of related data groups. We define *Joining* as the ability to allow the combination of data from different sources or tables based on common criteria, enabling data analysis in a broader context. We define *Sorting* as the ability to allow the sorting of data within a data-set according to a specified criteria order. We define *Pattern Matching* as the ability to to allow the combination of data from different sources or tables based on common criteria, enabling data analysis in a broader context.

3.6. Language

We define the *Language Feature* as the capability of the component to provide some language to manage and define queries over data. Different components can enable this feature through different categories of languages. For the purposes of the model we classify programming languages in declarative languages, imperative, object-oriented programming (OOP) languages, and functional languages. Declarative languages focus on the desired result: users specify what they want to achieve, and the system autonomously determines how to calculate and execute it. In imperative and object-oriented languages (OOP), the emphasis is on the operations to be performed to achieve the desired result; they differ in that imperative programming relies on a sequence of explicit instructions to

control program flow, while OOP encapsulates data and behavior within objects, promoting a more modular and reusable code structure. In functional languages, on the other hand, functions are treated as immutable entities that can be composed, allowing developers to define a processing flow in which functions accept arguments, perform specific operations, and return results based on specifications without causing side effects.

The feature in question is depicted in Figure 3.7. Components can implement it using one or a combination of various programming languages. In our model, a language feature can be provided by a component through the adoption of one or a combination of the following abilities: *declarative*, *imperative*, *OOP* and *functional*.

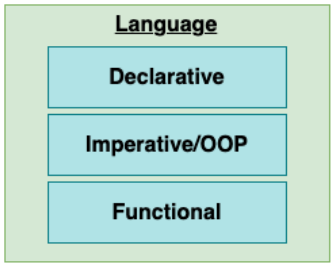


Figure 3.7: Language Feature.

We define *declarative language* as the ability of the component to provide the language capability following a declarative approach. We define *imperative language* as the ability of the component to provide the language capability following an imperative approach. We define *OOP language* as the ability of the component to provide the language capability following a OOP approach. We define *functional language* as the ability of the component to provide the language capability following a functional approach.

3.7. Extension

We define *Extensions Feature* as the ability of a component to adeptly manage and process data in alignment with their specific characteristics. These extensions encompass specialized algorithms and tools tailored to efficiently and accurately handle data. Indeed, the nature of data can, in certain instances, necessitate bespoke functionalities to ensure its proper management and analysis. Data can manifest in temporal and spatial dimensions, and may require to be considered accordingly. In the former, it can represent precise moments when events occurred (timestamps), time intervals, dates, hours, or any time-related information. In the latter, data can denote physical locations or spatial contexts where events unfold or objects reside. Furthermore, there are scenarios where data seam-

lessly intertwines both temporal and geographic attributes, and in these cases we speak about spatio-temporal data. A prime example is data originating from GPS sensors, which concurrently record geographic coordinates and corresponding timestamps.

In our model, we represent this feature as depicted in Figure 3.8, where we differentiate between *Temporal* and *Geospatial* abilities for managing and/or processing data.

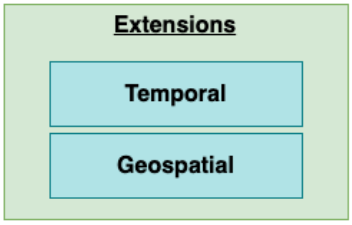


Figure 3.8: Extensions Feature.

We define *Temporal* as the capability to account for data within their temporal dimension, providing functions for managing this data effectively. These functions may include actions such as filtering, aggregating, or retrieving data based on specific time intervals. We define *Geospatial* as the capability to analyze data within their spatial dimension, offering functions to extract information based on geographic location. The architect may decide to select either one or both abilities. If both abilities are selected, spatio-temporal data is considered.

3.8. Parallelism

We define *Parallelism Feature* as the capability of the component to distribute data processing and/or storage across multiple nodes to improve performance. It entails parallelizing the execution of operations or queries whenever feasible, followed by aggregating the final outcomes. This minimizes the time needed for executing the entire operation. Both data and operations can be divided to facilitate parallel processing. In the first scenario, the data-set is partitioned into smaller segments, and the operation is concurrently executed on these distinct segments. In the second scenario, the query is dissected into independent tasks that are executed in parallel. Subsequently, the outcomes from these parallel executions are merged to produce the final solution. It's important to note that this approach might not always be feasible, as data dependencies could necessitate the system to execute sequentially in certain situations.

In our model, the feature is represented as in Figure 3.9.

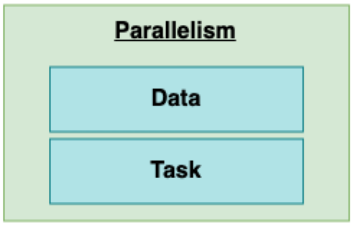


Figure 3.9: Parallelism Feature.

We distinguish between *Data Parallelism* and *Task Parallelism* abilities. We define *Data Parallelism* as the ability of the component to split the data-set to parallelize the operations, while *Task Parallelism* as the ability to split the query in tasks to be applied in parallel.

4 | Storage Features

This chapter describes the features each component can have when the storage is activated. We define *Storage Features* as capabilities the component's storage can exhibit, including the types of data it can store and the guarantees it provides regarding non-functional requirement.

Figure 4.1 provides an overview of the storage features in our model, all of which are present in the storage systems of each component type. Similar to computational features, storage features may demonstrate varying abilities when delivering each function, which are highlighted in the model.

Figure 4.1: Storage Features Overview.

The following section outlines the storage features that are related to both the requirements about the datatype to be persisted and the non-functional requirements.

4.1. Persisted Data Type Feature

We define *Persisted Data Type Feature* as the capability of a component's storage to persist specific types of data. Data can be categorized into three main types: structured, semi-structured, and unstructured. Structured data is organized with a predefined and coherent schema, typically represented in a tabular format where rows and columns are well defined. The schema for structured data must be defined in advance. Semi-structured data refers

to a type of structured data that doesn't adhere to the formal structure of data models commonly linked with relational databases or other types of tabular data. However, it still incorporates tags or other indicators to distinguish semantic elements and establish hierarchies of records and fields within the data. Unstructured data, pertains to data that lacks a predefined data model or doesn't neatly fit into relational tables. Unstructured information typically consists of a heavy amount of text, but it may also include elements such as dates, numbers, and factual data [50]. As a consequence, semi-structured data, has a more flexible schema with respect to the structured one. While it still exhibits some structure, it doesn't require a rigid, predefined schema. Semi-structured data often uses formats like JSON or XML and may include nested or hierarchical structures. The key distinction from structured data is that its schema doesn't need to be defined in advance, making it suitable for scenarios where data structures may evolve or vary. Unstructured data, in contrast, lacks a specific structure or format altogether. It doesn't conform to the tabular organization seen in structured data and doesn't follow a predefined schema like semi-structured data. Unstructured data can take many forms, including free-form text, images, videos, audio recordings, and more. Analyzing and extracting meaningful information from unstructured data can be challenging, as it doesn't have a consistent structure that makes it easily machine-readable.

The feature in our model is represented as in Figure 4.2. We distinguish between abilities in managing *Structured*, *Unstructured* and *Semi-Structured* data.

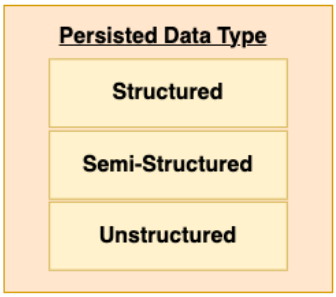


Figure 4.2: Persisted Data Type Feature.

We define *Structured Persisted Data Type* as the storage's ability to manage structured data. Similarly, we define *Unstructured Persisted Data Type* as the storage's ability to manage unstructured data, and *Semi-Structured Persisted Data Type* as the storage's ability to manage semi-structured data.

4.2. Availability

We define *Availability Feature* as the capability of the component to provide availability, which consists in being up and running whenever it is required by users. A highly-available system is defined as a system that is never blocked by some event, such as a failure[22]. This means that highly-available systems keep working even when part of the system goes down, giving the user the illusion that nothing has gone wrong.

Availability can be ensured by implementing replication. Replication is defined as keeping a copy of the same data on multiple machines, defined as replica, that are connected via network, allowing multiple entities to operate simultaneously on different replicas [37]. The replicated system should create the illusion for all users that they are operating within a centralized system rather than on different replicas. This concept is commonly defined as consistency among replicas. Different consistency models among replicas has been proposed that differentiate in the level of consistency they provide [58]. The strongest level of consistency is *linearizability* (or *strong consistency*), which guarantees that every operation appears to be applied instantaneously across all replicas. *Sequential consistency*, on the other hand, ensures that operations within each replica are executed in the same order as they were issued. In contrast, *weak consistency* does not guarantee that read operations return the most recently written value. Finally, *eventual consistency* is a level of consistency where replicas converge toward identical copies in the absence of further updates. This means that if no new operations are invoked on the object, eventually, all read operations will return the same value.

When implementing replication, systems require protocols to propagate updates from one replica to others. Systems can use either leader-based or leaderless solutions [37]. In Leader-Based replication, one (single-leader) or more (multi-leader) replicas are identified as Leaders, which are responsible for receiving all write operations and then propagating these writes to the associated replicas, which are known as followers. The propagation of updates, which can occur synchronously or asynchronously, compromises availability. With synchronous propagation, when a replica propagates an update, it waits until all recipients have processed it and sends an acknowledgment to signal completion. On the other hand, asynchronous propagation doesn't require the sender to ensure that the receiver receives the update and can continue processing other updates without being blocked. When a replica fails, leader-Based protocols implement logging on replicas. If the failed replica is not a leader, it recovers by using its log and contacting the leader to receive changes done when it was not active. If the failed replica is a leader, a failover mechanism has to be done to elect a new leader and inform other replicas about the change.

Therefore, in leader-based protocols, leader failover is costly, as it involves detecting leader failure, electing a new leader, and re-configuring the system to use the new leader. In case of synchronous propagation, this cause the system to block until the new leader is elected, and the cost is related to the time needed to restore the leader. However, in Multi-Leader protocols, this cost is restricted only by the subset of replicas associated with the failed leader, while the others can continue their operations without interruption. As a consequence, synchronous propagation impact in the level of availability, as it may require the system to be blocked, but do not impact on the level of consistency. In the case of asynchronous propagation, the failover cost is indeed related to the potential loss of data consistency guarantees. Since replicas process data as soon as they receive it without waiting for all nodes to have received it, if the leader fails before all other nodes have received the updated data, some replicas may not be aligned with the latest available data. This situation can persist until the failed leader is restored. When the leader comes back online, replicas will begin to converge toward a common and consistent state, but this process may take some time. As a consequence, asynchronous propagation do not impact on the level of availability, as do not require the system to be blocked, but provides only eventual consistency. The implementation of multi-leader systems introduces the challenge of write conflicts, where the same data can be concurrently modified in two different replicas [37]. This can occur due to factors such as asynchrony, where two leaders may receive different copies of data from their followers simultaneously. When these leaders propagate their changes, it can result in uncertainty among other nodes in the system regarding which version should be considered the latest and therefore prioritized. Consequently, it is essential to implement conflict resolution strategies to bring the system into a consistent state in such scenarios. These strategies can encompass both automated and manual approaches. In Leaderless replication, the producer sends writes to a number of replicas, defined as Quorum, which must approve that write before propagating it to all replicas. In other words, a quorum of replicas must approve a write operation before it is considered successful and propagated to all replicas. In the event of a replica failure, Leaderless replication ensures that all updates will eventually be executed through mechanisms such as read repair or anti-entropy processes. In the former approach, a client reads from multiple replicas concurrently and identifies divergent responses. In the latter case, internal processes within the component actively seek out discrepancies among the replicas and make the necessary adjustments. In this case, failover does not exist, and availability is ensured when a number of replicas greater than or equal to the quorum are active and functioning. This means that the system can continue to operate even if some replicas are unavailable or have experienced failures, provided that the required quorum is still reached. The quorum size can be manually configured based

on the specific needs of the system and the data. As a consequence, leaderless replication can be seen as a way to trade consistency and availability guarantees by selecting the quorum. The quorum algorithm is designed to ensure data durability and consistency, but it's important to note that there is a trade-off. Increasing the quorum size can enhance data durability and consistency but may result in increased latency and reduced system availability. Furthermore, by requiring approval from multiple replicas through the quorum, it's possible to reduce write conflicts. Each write operation is processed by more than one replica, increasing the chances of effective conflict resolution.

Based on this, we categorize protocols into *blocking* and *non-blocking* categories. We classify protocols as *blocking* when they may necessitate communication to be temporarily halted, whereas we label protocols as *non-blocking* when they allow the system to continue functioning even in the event of replica failures. The first category, comprising protocols that propagate updates synchronously, prioritizes strong consistency guarantees but tends to be less available. This is because they may require system suspension in the event of a replica failure and often involve the implementation of failover mechanisms. Conversely, non-blocking protocols, which propagate updates asynchronously, are highly available, as they can continue operating even when replica failures occur. However, they typically provide only eventual consistency guarantees. Leaderless replication is considered a non-blocking alternative that offers robust consistency guarantees, assuming that the concurrent failure of replicas remains below a certain quorum threshold.

The *Availability Feature* of our model is shown in Figure 4.3. We distinguish between the ability of the storage's component to be *Highly-Available* or *Available*.

Figure 4.3: Availability Feature.

We define *Higlhy-Avaliable* as the ability to ensure high-availability, achieved by a component implementing replication using a blocking protocol. Conversely, we define *Availability* as the ability to be available, which is accomplished by a component implementing replication with a non-blocking protocol alongside appropriate failover mechanisms. The figure also highlights the trade-off involving consistency guarantees: achieving highly-available abilities typically implies offering only eventual consistency guarantees, whereas

availability can allow us to attain strong consistency guarantees.

4.3. Atomicity

We define *Atomicity Feature* as the capability of the storage's component to ensure atomicity among different transactions. We consider a transaction as a group of operations that can be reads, writes and/or updates on multiple items. Atomicity ensures that all transactions are seen as logical units, meaning that either all operations composing the transactions entire succeed, or the entire transaction fails. The first case is defined as transaction commit, while the second is defined as transaction abort [37].

To ensure atomicity in distributed settings, the 2-Phase Commit (2PC) can be implemented. It ensures that operations either commit in all nodes or abort[37]. This is done by reaching an agreement between the replicas, which can be a global-commit or a global-abort. This protocol requires the existence of a coordinator who manages the nodes: when the transaction finishes, if there is a commit request by the application, the coordinator emits the vote-request message, which is sent to all the replicas; replicas receives the vote request and vote commit or abort; the coordinator collects all votes and if there is at least one abort emits a global-abort to replicas, otherwise it emits a global-commit; replicas receive the global message and move to the corresponding state. This protocol works under the assumption that the coordinator never fails, otherwise execution might be blocked until the coordinator recovers: it is a blocking protocol. A more expensive protocol that solves this problem is 3-Phase Commit (3PC), which avoids blocking the execution in case of coordinator failure. However, in practice it is preferable to use the less expensive 2PC and implement some failover mechanism[37]; for this reason, 3PC is omitted from the model. Summing up, Two-Phase Commit (2PC) is a valid solution for ensuring atomicity in distributed systems, as it employs a voting mechanism to ensure that all replicas agree on whether to commit or abort a transaction, thus achieving atomicity. However, it comes with the drawback of needing a failover mechanism, which can potentially impact system availability if not managed effectively.

To increase system availability, it is possible to relax the requirement of atomicity and implement non-blocking solutions. Sagas have been introduced in the context of centralized systems, but they are applicable to distributed systems too [30]. The concept behind sagas is to split the transaction in a sequence of independent step where each step does not have to observe the same consistent state. Each step update the storage and trigger the execution of the next one. If a step fails, compensating actions are performed to undo the changes done by the previous one. The idea behind sagas is to allow intermediate

inconsistent states of the storage, by assuming that it is always possible to asynchronously rollback without blocking the execution. For example, consider a transaction composed of hotel booking and flight booking operations and assume that there is an integrity constraint that requires only both to be booked: the transaction is successful only if both operations occur; if only one of them fails, both are canceled. If the first operation is successful and the second is not (for example because there are no flights available), using the sagas it is possible to temporarily commit the first operation, and only after having received the abort of the second to automatically roll back the first. However, immediately after the first operation was completed, the storage was in a state in which the user had only booked the hotel, violating the integrity rule. In this case, it can be considered acceptable because it only affects the user, for whom seeing the intermediate state is difficult because he would have to log in with another system at the same time. So assuming the user is logged in with only one device at a time, the sagas are acceptable.

Another way to handle the execution of operations in a transaction relies on a deterministic approach: the order is deterministically established and delivered to all replicas before transactions are executed. It uses a distributed log, where operations are written in a serial order; each replica has a local copy of this log that is used to execute operations. In this way, all nodes execute operations in the same order, which is the one in the log. This approach simplifies the commit protocol, as in the event of a machine failure, it is sufficient to re-execute the transactions recorded in the distributed transaction log. Consequently, in this scenario, the two phases of the commit protocol become unnecessary, thereby reducing the need for additional communication and rendering the protocol non-blocking precisely when locks are detected[56].

The feature in our model is represented as in Figure 4.4. We distinguish between the ability of the storage's component to provide *Global Transactional* guarantees, *Eventual Transactional* guarantees or *Local Transactional* guarantees.

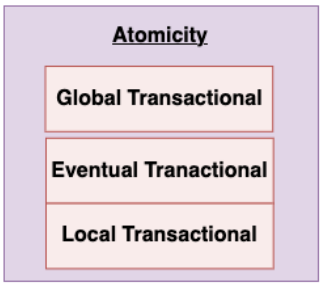


Figure 4.4: Atomicity Feature.

We define as *Global Transactional* guarantees the ability of the component to implement a blocking-protocol to provide atomicity guarantees over distributed transactions, such as 2PC. We define as *Eventual Transactional* guarantees the ability of the component to allow for some strategy to provide atomicity guarantees that eventually the transaction will be seen as atomic, but accepting temporarily incorrectness. We define as *Local Transactional* guarantees the ability of the component to allow for some strategy to provide atomicity guarantees over local transactions, such as Deterministic approach.

4.4. Isolation

We define *Isolation Feature* as the capability of the storage's component to provide isolation among different transactions. This is useful when replication is implemented to ensure consistency among replicas[18]. We consider a transaction as a group of operations that can be reads, writes and/or updates on multiple items. Isolation is defined as the property of ensuring that concurrently executing transactions do not affect each other; if, for example, one transaction makes several writes, then other transactions should see either all or none of them [37]. Transactions that reads data that is concurrently modified by another transaction, or two transactions that try to simultaneously modify the same data, may cause concurrency issues may arise, as race conditions, which impact the level of consistency among replicas. Examples of race conditions include dirty read, which occurs when a transaction reads an update made by another transaction that has not yet committed and may be aborted, non-repeatable read, which occurs when a data item is read twice within the same transaction with different values, and phantom read, which occurs when a transaction executes two different queries in the same processing and returns two different results[18]. Hence, to avoid this, concurrency control techniques must be implemented.

The strongest level of isolation, defined as *Serializability*, ensures that transactions, even when executed in parallel, produce the same final result as if they were executed sequentially, one after the other, without any concurrency. This means that if transactions are correct when executed individually, they will still be correct when executed concurrently [37]. To ensure serializability, a first requirement is a global transactional atomicity ability of the component (Subsection 4.3): transactions must either entirely commit or abort. Moreover, a lock-based protocol can be implemented. These mechanisms use locks to allow one transaction at a time to access the data. One of the most common lock-based protocol is the 2-Phase Locking algorithm (2PL). However, implementing some protocol to ensure this isolation level, especially in distributed scenario, may bring to reduce

availability, which is undesirable in many applications. For example, in a distributed environment, performing 2PL for a transaction of length T necessitates T lock operations along with at least one lock and one unlock operation, each of which demands coordination with other database servers or a lock service[22]. Moreover, they can cause deadlocks that are usually solved automatically with some technique by the system[37]. Another way to ensure serializability is to use a Multi-Versioning Concurrency Control (MVCC) protocol with a timestamp implementation, where resources are tagged with the timestamp of the transaction that created them, and read operations access the correct version based on the timestamp. However, it is still necessary to check for update conflicts to manage situations where concurrent writes can occur. This check makes the protocol blocking and unsuitable for highly available solutions[22].

Consequently, various weaker isolation levels have been introduced and categorized based on the race conditions they mitigate or avoid [18], and some of them can be implemented in highly-available scenarios[23].

For example, snapshot isolation ensure that each transaction sees all the data that was committed in the database at the start of the transaction, by reading from a consistent snapshot of the database[37]. However, this approach necessitates the implementation of locking mechanisms to prevent dirty writes, making it unsuitable for high availability scenarios [22]. Nevertheless, it doesn't require any locks for read operations, enabling the database to concurrently execute long-running read queries on a consistent snapshot while processing writes in a typical manner, without encountering lock contention between the two. In many scenarios, this level of availability suffices, rendering it a commonly employed solution[37].

An example of isolation level that can be implemented with highly-availability is the Read Committed isolation level[23], ensure transactions to read only committed values, prevents dirty reads but not non-repeatable reads. This is implemented by most systems with a MVCC approach by retaining both the old committed value and the new value set by the current transaction holding the lock for writing. Other transactions reading the object during this transaction's execution access the old value, transitioning to the new value only upon its commitment[37].

Hence, it has been demonstrated that many systems can achieve highly-availability by relaxing serializability guarantees and allowing some race conditions to occur[23]. This is acceptable in many applications where absolute data consistency is not a critical priority, and a certain level of ambiguity in results can be tolerated.

By generalizing, we can conclude that there are blocking protocols that offer robust isola-

tion guarantees, such as serializability, but are not suitable for components with stringent availability requirements. On the other hand, non-blocking protocols provide weaker isolation levels, like read committed, but are well-suited for components that prioritize high availability. Snapshot isolation falls in between these extremes, as it strikes a balance by delivering both high availability and strong consistency, particularly when handling long-running read queries.

Figure 4.5 shows the feature's representation in our model. We distinguish between the component's ability to ensure *Strong* isolation guarantees and *Weak* isolation guarantees.

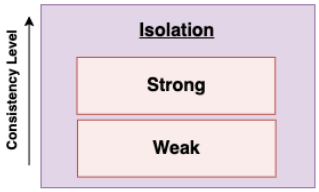


Figure 4.5: Isolation Feature.

We define *Strong Isolation* as the ability to offer strong isolation guarantees, typically enforced through a blocking protocol. However, this approach comes at the cost of reduced availability. Conversely, we define *Weak Isolation* as the ability to provide weaker isolation guarantees. This is implemented with a non blocking protocol, often implemented via a non-blocking protocol, thus enabling higher availability. The feature also highlights the direct correlation between the strength of the isolation level and the degree of data consistency it ensures.

4.5. Durability

We define *Durability Feature* as the capability of the storage's component to provide durability guarantees. Durability is defined as the promise that once a transaction has committed successfully, an data it has has written will not be forgotten, even if there is a hardware fault or the database crashes. In a replicated database, durability implies that data has been successfully copied to a certain number of nodes. To ensure durability, a database must wait until these write operations or replications are completed before confirming a transaction as successfully committed[37]. Different techniques can be implemented to provide durability guarantees.

Checkpointing techniques are used to save the state of the system to a stable storage in order to use it in future when necessary. There are two different mechanism to imple-

ment it: Independent Checkpointing and Distributed Checkpointining. With Independent Checkpointing nodes without coordinating periodically checkpoints and save their state independently one to each other. This mechanism is easy to be implemented, but it could end up in an inconsistent set of checkpoints: it can happen that a node has checked pointed up to an event X and then started an event Y, and another node at the moment of the checkpoint has already received Y. This cause the system to search for another set of checkpoints which is consistent, going back in time: the first consistent cut found going back represents the recovery line, which is the sets of checkpoints that together result consistent. This may lead to the domino effect: if no checkpoint represents a consistent cut, the system must come back to the initial state of each node. With Distributed Checkpointing the system takes a distributed snapshot that represents the state of the application and save it on the disk. The advantage here is that processes are not blocked while the checkpoint is being take; the drawback is the increased complexity of the global snapshot algorithm to be implemented. Another difference is that with Independent Checkpointing, that does not require coordination, previous checkpoints have to be persisted to find the first consistent cut, whereas Distributed Checkpointing, that require coordination, does not need to save previous snapshots.

Logging techniques can be used to store the state of the system to a stable storage too. The difference is that instead of saving the entire state, the Logging feature saves either the input data and invocations on it to restore the state, or saves state changes before updating the main storage. The first mechanism is defined as Command Logging, whereas the second as Write Ahead Log. Differently from write ahead log, Command Logging requires additional processing, but it is faster; on the other side, with a Write Ahead Log all changed data are persisted.

These two techniques are commonly combined approach is to start from the last checkpoint and re-execute the transactions recorded in the log.

Figure 4.6 shows the feature's representation in our model

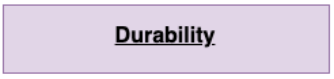


Figure 4.6: Durability Feature.

4.6. Scalability

We define *Scalability Feature* as the capability of the storage's component to provide scalability as the data volume grows. Scalability is defined as the system's ability to cope with increased load[37]. System scaling refers to the capability of a system to tolerate an increase in demands for data processing[20].

Scalability is divided in two main categories: Vertical and Horizontal Scalability. To accommodate the growing data volume, a vertically scalable approach involves enhancing the resources of a single node, while a horizontally scalable approach entails adding another node with equivalent resource capacity[37].

Vertical scalability typically involves the installation of additional processors, faster hardware and memory into a single server[20]. The problem with this approach is that the cost grows faster than linearly. Horizontal scalability, on the other hand involves workload distribution over several servers (these may even be commodity machines)[20]. In other worlds, it involve splitting the data-set in smaller partitions and store them independently, enabling the addition of new partitions as the data volume expands.

Figure 4.7 shows the feature's representation in our model, where we distinguish between component's abilities of providing *Horizontal Scalability* or *Vertical Scalability*

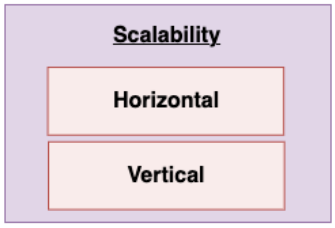


Figure 4.7: Scalability Feature.

5 | Physical Components

In this chapter, we provide categorizations for the systems used in building distributed infrastructures. Through a thorough examination of both literature and official documentation for various systems, we have categorized them based on the capabilities they cover in our model and the approaches they employ. In other words, we have delineated the abilities that different categories of systems offer.

In accordance with what we have done with the components of our model, we categorize systems into two main categories: state-centric and data-centric, both with the goal of optimizing the utilization of large-scale data, but differing in their primary focus. We define *data-centric* systems as those designed for data processing, emphasizing their efficient processing, analysis, and manipulation. We define *state-centric* systems as those that primarily revolve around data management, including their storage and organization. While some of these systems may include computational capabilities, their primary purpose is to ensure the effective maintenance of data state, often in distributed environments with large data volumes to manage.

This categorization is derived from an in-depth review of literature and official system documentation, with the aim of identifying patterns that can be leveraged for the methodology. After careful analysis, we can assert that distributed systems within the same category generally behave in a similar manner, with some secondary differences. Therefore, for the methodology's purposes, we have highlighted the key characteristics within our models that distinguish them and guide the choice of the category of distributed systems to use for implementing architecture components. The other characteristics, referred to as secondary, are used to differentiate between different systems within the same category. However, given the wide variety of systems available today, in the subsequent sections of this chapter, we provide a general overview of the categories, followed by some notable examples of similarities and differences. We define as primary features all storage features and Data Acquisition, Processing Request and Processing Response Computation Feature.

The methodology will provide a set of technology categories to use, and the architect

can then choose the specific system based on both the secondary characteristics and their practical experience. We have made this choice to allow architects to select individual systems they are more familiar with.

In the following sections of the chapter, we provide a general overview of the different types of systems, both state-centric and data-centric. We illustrate how these systems align with our model by specifying the features they cover and how they cover them. It is worth noting that data-centric systems do not provide any storage features, as they are not designed to maintain state within the infrastructure, while state-centric systems may, to a limited extent, offer computational capabilities.

Section 5.1 addresses data-centric systems, while Section 5.2 delves into state-centric systems. Both sections present the primary categorizations of these systems and briefly discuss those that fall outside the main categories. Additionally, they demonstrate how the various categories fit within our model.

5.1. Data-Centric Systems

This section delves into systems tailored for data-centric components, which we term Processing Systems. Firstly, we categorize them, highlighting their similarities and differences. Then, we delve into a detailed analysis of their characteristics to demonstrate how these systems align with our model.

In the remainder of the section, in Subsection 5.1.1, we introduce the different Processing Systems, and in subsection 5.1.2, we demonstrate their mapping within our model.

5.1.1. Processing Systems

We define *Processing* Systems those systems whose characteristics primarily focus on performing computations over data without persisting it over time. They require the definition of a pipeline of actions to be performed over the data. We categorize them into *Batch-Processing* systems and *Stream-Processing* Systems, with the former implementing batch processing and the latter executing stream processing. A stream-processing system processes data streams as the data arrives. It aggregates, filters and analyzes the data items and thus gains fast insights, reactions to observed situations and higher level information [49]. Batch processing is a data processing model in which information is collected and aggregated into a predefined 'batch' before processing, whereas real-time processing entails the ongoing processing of data as it arrives [26]. A processing system can belong to both categories if it has the capability to perform both batch and stream processing

tasks, and we refer to such systems as *Hybrid-Processing* systems. Subsection 5.1.1 analyzes the two categories, highlighting differences and commonalities, while Subsection **??** demonstrates how various processing systems align with our model.

Stream-Processing Systems vs Batch-Processing Systems

As mentioned earlier, the main difference between batch-processing systems and stream-processing systems lies in how they handle data. Batch-processing systems focus on large volumes of data accumulated over time, while stream-processing systems focus on real-time data streams. Consequently, batch-processing systems are better suited for complex processing tasks, while stream-processing systems are ideal for lightweight and fast tasks such as filtering and aggregation in real-time. An optimal use case for batch-processing systems is periodic analysis, while stream-processing systems are suitable for system monitoring. Another technological difference that influences the choice is the data arrival mode. We sow that data can enter the system either continuously, one item at a time, or periodically as a group of items. When data enters continuously, both batch-processing and stream-processing systems can handle it. Streaming technology systems handle data items one at a time as soon as they arrive, while batch-processing systems gather and process a group of items once a batch is collected. If data enters the system periodically, only batch-processing systems can process it, by performing the computation as soon as the group of items arrive. As a result, batch processing can interact to receive data with both external producers and other internal components, including batch processing and stream processing. Additionally, batch processing can also fetch data from a database in groups for processing, allowing it to interact for data reception with a state-centric component as well. Stream-processing systems instead require data to arrive in a streaming manner, so they need to be connected either to external sources or other stream-processing systems. Consequently, when data needs to be fetched from storage, stream-processing systems are not the appropriate choice. Another significant difference is that batch processing has higher latency, while stream processing offers much lower latency, as in the first case, data is processed immediately, whereas in the second case, it takes more time both to collect the data and to execute a more complex process [26]. However, stream technologies are much more costly to implement because of their real-time processing requirements (low latency, rapid data ingestion, rapid response to event). Also, they require a more robust infrastructure to handle the contonous stream management.

Summing up the choice between the two approaches will depend on the nature of the data, latency requirements, specific applications, and the complexity of processing tasks.

In practice, a combination of both methods is often used depending on the needs.

Among the prominent stream processing systems, we highlight Apache Kafka Streams[9], , Apache Storm[16]. For batch processing, we have Hadoop with MapReduce[7]. Also, Spark[15] and Apache Flink[5] provide both stream and batch processing capabilities.

5.1.2. Components Mappings

In the following sections, we provide the mapping of different categories of processing systems within our model. As mentioned earlier, these systems are primarily focused on computation and do not involve persistent data storage. Consequently, storage features are not provided by these components. Therefore, we will analyze them solely in relation to computation features.

Computation Feature Mapping

Data Acquisition In Section 3.1 we have defined *data acquisition* as the component's ability to obtain data from either other components or producers. We have made a distinction between *Single-Item Data Acquisition*, where one item is inserted into the component at a time, and *Multi-Item Data Acquisition*, where a group of items is inserted simultaneously. In processing systems elaboration entails executing actions on items, either individually or in groups, as soon as they are acquired. Stream-processing systems elaborate one item at a time, and as a consequence they have a single-item data acquisition by default. Conversely, in a batch-processing system, the elaboration of single items is atypical, and hence single item data acquisition too. Typically, data is elaborated in batches, where multiple elements are elaborated simultaneously within each batch. As a consequence, the most frequent approach for batch-processing systems is multi-item data acquisition Moreover, stream-processing systems can perform multi-processing data acquisition. In fact, they implement windowing techniques to partition the data stream into batches and perform batch data processing, where a batch of stream elements for a specific stream represents a finite subset of the stream within a time interval[36]. In the case of multi-item elaboration, processing systems typically incorporate some form of temporary storage mechanism to hold the group during the execution of operations. Windowing operators implemented by stream-processing systems are defined as stateful operators, where the state can be defined as the intermediate value of a specific computation that will be used in subsequent operations during the processing of a data flow[57]. Also, batch-processing systems when receives data from a stream, uses some state to group items before starting the elaboration. This should not be confused with the storage

of our model. In fact, to enable this capability, the architect is not obligated to activate storage explicitly, but by enabling multi-item ability in a data-centric component, it implicitly requires the processing system to possess some form of temporary storage capability. This highlight a gray area in the model: in a sense, a stateful technique could be considered as a database, since it persists information. However, at the moment these are separate concepts: there are technologies that are not databases but that can be based on a state. These technologies in the model are mapped to Data-Centric components with multi-item elaboration ability activated. Notice that recent researches try to converge these two worlds[19], trying to implement systems that converge processing and storage, but at the moment they are only prototypes. Stateful mechanisms allow one to perform windowing, which simplify some types of computation on stateful processing in which the state does not have to be maintained interactively, but declaratively there are systems that allow to define which portion of data is being acted on. Moreover, they allow, during the Processing Phase, to perform Incremental Processing, which is a processing technique where the new state is calculated incrementally by combining the old state with new data, reducing processing latency.

Processing Request and Query Knowledge We have defined in Section 3.2 *processing request* as the capability of the component to receive a query that enters the architecture. We have defined *static* queries as queries known and established during the architecture implementation, primarily focused on data processing task, and *dynamic* queries as those ingested by users during the system's usage. Moreover, we have defined in Section 3.4 *Query Knowledge* as the component's capability to manage queries that may be known at various points in time. We have distinguished between *known queries* as those queries known in advance at the time of the architecture implementation, and *unknown queries* as those queries that are not known or predefined in advance but they may be ingested in the system at real time. The design of data processing systems entails the management of static queries, which are carefully planned in advance to fulfill specific system requirements. Because the sequence of actions to be executed by processing systems must be predetermined, they are incapable of handling unknown queries. Furthermore, these systems cannot accommodate dynamic queries because they do not retain their results. To handle unknown queries, they must depend on a storage mechanism. Consequently, they are primarily suited for addressing predetermined, static queries. In general, processing systems are ill-suited for managing dynamic queries due to their lack of result persistence.

Processing Response We have defined in Section 3.3 *Processing Response* as the capability of the component to send out data requested either by the consumer of by another component . We have distinguished between the capability to respond *periodically* to requests and the capability to provide *continous* responses. As they are defined, batch processing systems respond periodically to queries, while stream processing systems handle data in real-time or near-real-time, providing instant responses upon data arrival.

Language We have defined in Section 3.6 the language features as the capability of the components to provide some language to manage and define queries over data. Looking at the documentations of processing systems, we can conclude that the most of them are designed to be compatible with a wide range of general-purpose programming languages, offering adaptable APIs and libraries for various languages, thus enabling the implementation of big data processing applications in different languages. These languages can be both imperative and OOP. Furthermore, processing systems have the flexibility to integrate functional or declarative paradigms. In stream-processing systems, APIs tend to have a more functional nature, with data transformation operators (`map`, `filter`, etc.) often used in a functional manner on data streams. For example, in Spark, we can define data transformation operations on a data stream in a functional manner using the DataFrames API, while Flink promotes the use of functional operations on real-time data streams. On the other hand, batch-processing systems provide APIs with a more declarative approach, similar to a query language used in databases. For instance, Spark offers the DataFrame API, which allows data manipulation similar to SQL, thus introducing a declarative paradigm for batch data processing.

Extensions We have defined in Section 3.7 *Extensions Feature* as the ability of a component to adeptly manage and process data in alignment with their specific characteristics. Both batch-processing and stream-processing systems often incorporate extensions to handle temporal and spatial data. In batch-processing systems, extensions for temporal data frequently include libraries or functions designed to manage time series data, enabling efficient time-based processing. For example, Apache Spark offers the Spark SQL module, which supports temporal data types and date/time functions. It also allows joining data based on temporal columns and partitioning data based on time. Extensions for spatial data in batch-processing systems can provide support for geographic data types and geospatial query functions. They may also include algorithms for performing spatial joins, calculating distances, and aggregating spatial data. For instance, Spark offers the GeoSpark extension, specifically designed for geospatial data analysis. GeoSpark allows to define geographic coordinates and use specialized functions for spatial data, such as

distance calculations. In the case of stream-processing systems, extensions for temporal data are often necessary to support the processing of data streams with timestamps. However, for detailed temporal analysis, it is often necessary to use time windows that allow data to be grouped into specific time intervals. Spark Streaming, for example, provides time-based windowing functions for this purpose. Extensions for spatial stream-processing systems can offer functionalities for detecting spatial patterns, tracking moving objects, or performing real-time geofencing operations. The GeoMesa extension introduced for database systems, can also be integrated with Spark Streaming to perform real-time spatial analysis.

Parallelism In Section 3.8, we have defined parallelism as the capability of the component to distribute data processing across multiple nodes to improve performance. In general, data processing systems have as primarily task to process data efficiently, and parallelism is a fundamental component to achieve this objective. The ability to perform operations in parallel on data is essential for enhancing performance and speeding up data processing[44] In batch-processing systems, we consider data parallelism as the parallel processing of batches, while task parallelism as the simultaneous execution of various independent operations on each batch. In stream-processing systems, we consider data parallelism the execution of multiple instances of an operator, in parallel on different parts of the input data. Task parallelism, on the other hand, involves the execution of multiple operations on the same input stream in parallel[49]. In data processing systems, both those based on batch processing and those oriented towards streaming, the management of parallelism can vary significantly and can be defined by the user or automated by the system. In general, in batch processing systems, automation of parallelism is more common, as batch processing involves the processing of data-sets in separate batches. For stream processing systems data parallelizing statefull operators requires much more attention when defining the pipeline of execution, because the input stream should be distributed among the operator instances such that each instance can keep an individual state to avoid interference in between the different operator instances[49].

Processing Actions In Section 3.8, we have defined processing actions as those actions used to extract derived information. Upon reviewing the documentation for various processing systems, we can state that that both batch-processing and stream-processing systems are equipped to perform data processing tasks. This aligns with the state-centric systems definition we have provided, in which we categorize processing systems. In fact, we define state-centric systems as those whose primary objective is to process data. Batch-processing systems allow for the filtering of elements within a batch based on spe-

cific conditions, aggregating batches to generate statistics or aggregated results, grouping or merging data based on common characteristics, and performing pattern matching to search for specific sequences or patterns in the data, such as keywords in text. In stream-processing systems, filtering pertains to the selection of incoming data based on specific conditions. Aggregation and grouping are accomplished using stateful techniques. Joins involve merging data from different streams based on common keys or specific conditions. Pattern matching serves to identify specific real-time events, for instance, for the purpose of detecting anomalies within a stream.

5.2. State-Centric Systems

In this section, we categorize state-centric systems and begin by providing an overarching overview of them. Subsequently, we delve into analyzing their characteristics and how they relate to our model. We classify state-centric systems as those systems whose primary objective is the storage of large volumes of data. As the primary category of State-Centric Systems, we include *Database Systems*, which we define as systems dedicated to efficiently and coherently organizing, storing, retrieving, and manipulating data. Databases can be accessed by multiple users or applications, providing mechanisms to ensure concurrency, availability, and reliability. By concurrency, we mean the ability to access data concurrently by different users; by availability, we mean the system's ability to respond to requests; and by reliability, we mean the system's ability to recover data in case a part of the system fails. Therefore, databases fall into the category of state-centric systems capable of providing the capabilities of our model related to storage, i.e., storage features.

As another category of state-centric systems, we identify *Distributed File Systems(DFS)*. These systems are also primarily designed for the storage and distribution of files and data within a distributed network[39]. They offer the capability to store large amounts of unstructured data in forms of files reliably on commodity hardware.

In the remainder of the section, in Subsection 5.2.1, we introduce the Database Systems, in subsection 5.2.2, we delve into other types of databases, and finally, in subsection 5.2.3, we demonstrate the mapping of state-centric systems within our model.

5.2.1. Database Systems

In this section, we identify the two main categories of database systems, which are SQL and NoSQL database systems, and we provide a general overview of their main charac-

teristics.

SQL solutions are relational databases where data is stored and manipulated using a predefined schema that must be established in advance. This schema is defined through tables, which can be interconnected. These solutions ensure ACID properties: they offer strong guarantees in terms of Availability, Consistency, Isolation, and Durability[25]. Consistency refers to Integrity: the SQL language is used for implementing transactions on the data while maintaining data integrity between each transaction. However, they exhibit limitations in terms of flexibility, scalability, and performance as data volume grows. In fact, SQL solutions are primarily vertically scalable, and they lack flexibility since the schema must be determined in advance and cannot be easily altered.

NoSQL databases are the development of horizontally scalable, distributed non-relational data stores [43]. They offer enhanced flexibility due to their dynamic schema, allowing the storage of unstructured and semi-structured data. Additionally, in contrast to SQL databases, they provide horizontal scalability. According to the distinction we provide in Section 4.6, as data volume grows, the economic expense required is lower.

Different NoSQL solutions are available, which differs in the way they persist data and in the query expressiveness they offer[32][38][43].

- *Document-based* NoSQL solutions stores data as documents, usually in formats such as JSON or BSON. Their queries offer a high degree of expressiveness, enabling filtering, manipulation, and data aggregation. It's important to note that to fully leverage this expressiveness, the document schema should be thoughtfully designed. They allow for indexing documents based on attributes present in the document, improving search queries not only focusing on the primary key.

- *Key-Value* NoSQL solutions associate each item with a key and store data as key-value pairs. Their queries are primarily based on direct lookups and do not provide the ability to filter or aggregate data.

- *Graph-based* NoSQL solutions store data in graph structures, where nodes represent items, and edges represent the relationships between them. Their queries primarily focus on traversing graphs to explore data relationships, rather than on performing joins or aggregations.

- *Column-based* NoSQL solutions store data in a columnar format. Consequently, they may necessitate the upfront definition of a schema. Nevertheless, they offer greater flexibility compared to SQL databases as they permit dynamic column definition. Additionally, columns can accommodate text or JSON to store unstructured data,

although this may not always be the optimal approach. Their queries provide a high level of expressiveness, enabling operations such as sorting, joining, aggregations, filtering, and scans, all optimized for columnar data. Their queries are concurrent process queries, that is, each column is treated by one process. However, they are less structured than relational databases as they do not support table association.

Based on this, various NoSQL database types are better suited for different query scenarios[43]. For simple lookup queries, key-value solutions excel: their query speed is higher than relational databases, support mass storage, and high concurrency[32]. For intricate searches relying on specific attributes, document-based databases are preferable. When the queries involve the analysis of complex relationships among items, graph databases become the choice of consideration[38]. Lastly, for queries primarily focused on analytics and aggregations, column-based databases emerge as the optimal selection[32].

As will be further detailed later, databases differ in how they handle the CAP Theorem, which asserts that any networked shared-data system can have only two of three desirable properties among Consistency, Availability, and Partition Tolerance. [25].SQL databases focus on consistency and availability, using replication techniques to provide it and thus not providing guarantees of partition tolerance[32]. NoSQL systems, being horizontally scalable, choose between Availability and Consistency: some of them ensure consistency among the different nodes but are less good for scalability, while others are better in terms of scalability but they lack in consistency.

In the following, we describe how SQL and NoSQL database systems map to our model. To get a complete mapping, we reviewed documentations from many SQL and NoSQL databases. In this chapter, we provide examples about MySQL[10], Oracle[12], PostgreSQL[14] for SQL databases, MongoDB[3], CouchDB[2], Cassandra[1], Redis[4], Neo4J, HBase for NoSQL.

5.2.2. Others State-Centric Systems

A file system is a subsystem of an operating system whose purpose is to provide long-term storage. [39] A file system serves as a fundamental component of computer storage, responsible for managing and organizing files in a structured manner. Its primary objective is to provide a reliable and durable means of storing data for the long term. A distributed file system (DFS) extends this concept to operate in a distributed or networked environment. It is spread across multiple servers or nodes, creating a networked architecture[39]. So, just like databases, DFSs are used for data management, allowing the storage, organization, and retrieval of information. They are designed to ensure the reliability and

durability of data through mechanisms such as data replication or backup[55]. However, they primarily handle unstructured or semi-structured data, such as files and documents, with access typically based on files and directories, and they do not support complex queries like databases. Also, they do not support ACID transactions and lack consistency guarantees like databases. In fact, they are designed to prioritize availability by utilizing file replication for redundancy and availability. However, in many cases, they may sacrifice availability to maintain the consistency of critical files[39]. In file systems, fault tolerance is closely tied to the replication feature because replication is implemented to ensure availability and to transparently handle failures for users[55]. Furthermore, they are designed to prioritize availability, sometimes at the cost of consistency. They utilize file replication for redundancy and availability. In fact, they do not support ACID transactions and lack consistency guarantees like databases. However, in many cases, they may sacrifice availability to maintain the consistency of critical files[39]. Furthermore, they are designed to allow multiple users or applications to access data simultaneously, supporting concurrency. They can be scaled to meet growing data storage and access needs[39].

So, in general, we can say that DFS (Distributed File Systems) are often an appropriate choice when you want to store data without the need for strict structuring or complex querying operations. DFS are particularly well-suited for storing unstructured or semi-structured data, such as text files, images, videos, and documents, and they are designed to provide a scalable and reliable storage mechanism for such data. They are commonly used in conjunction with data processing systems, where data is stored in a DFS for access by big data frameworks like Apache Hadoop [51]for processing and analysis.

5.2.3. Components Mappings

Computation Components Mapping

Data Acquisition In Section **??** we have defined data acquisition as the component's ability to obtain data from either other components or producers. We have made a distinction between *Single-Item Data Acquistion*, where one item is inserted into the component at a time, and *Multi-Item Data Acquisition*, here a group of items is inserted simultaneously. All databases provide strategies to perform both single-item and multi-item data acquisition. SQL databases allows with the SQL clause INSERT to insert either a row or a set of row at a time. Similarly, document-based databases offer flexibility when it comes to data insertion. For instance, MongoDB, with its query language, provides methods like `insertOne` for inserting individual documents and `insertMany`for adding multiple documents all at once. Some key-value allows to insert both single-item and

multi-item. For example, Redis allows for both single-item and multi-item insertion, while Amazon DynamoDB does not support the insertion of multiple items in a single operation through a single statement. Instead, you need to execute separate insert operations for each individual item you wish to add to the database. Similar to key-value databases, there are variations among column-based databases in their support for multi-item insertion. For instance, Apache Cassandra, a column-based database, allows the insertion of multiple records concurrently through batch operations. In contrast, Base, another column-based database, does not support multi-item insertion through a single batch operation. Graph-based databases have the capability to handle both single-element and multi-element insertions. For example, Neo4j, which is a graph-based database, allows the insertion of multiple nodes and edges in a single transaction. You can add multiple elements concurrently by using a single query for creating nodes and relationships in Neo4j.

Processing Request and Query Knowledge We have defined in Section 3.2 *processing request* as the capability of the component to receive a query that enters the architecture. We have defined *static* queries as queries known and established during the architecture implementation, primarily focused on data processing task, and dynamic queries as those ingested by users during the system's usage. Moreover, we have defined Section 3.4 *Query Knowledge* as the component's capability to manage queries that may be known at various points in time. Moreover, we have defined *known queries* as those queries known in advance at the time of the implementation, and *unknown queries* as those queries that are not known or predefined in advance. In general, databases can respond to queries at any time, provided that the query pertains to the data stored within them. However, the ability to respond to a query can vary depending on the type of database and the data structure in use. Some databases, especially those based on rigid schemas, may struggle to effectively handle queries that have not been planned in advance. For instance, we have seen that relational databases require queries to be written in SQL and the table structure to be defined in advance. This can limit their flexibility when it comes to handling dynamic or unforeseen queries. On the other hand, NoSQL databases are designed to be more flexible and can handle dynamic queries more easily. For example, document-based databases are less strict regarding data schema, allowing for dynamic data insertion and query definition, even if these data and queries were not known prior to the data insertion.

Processing Response We have defined in Section 3.3 *Processing Response* as the capability of the component to send out data requested either by the consumer of by another component . We have distinguished between the capability to respond *periodically* to

requests and the capability to provide *continous* responses. Databases are primarily designed to respond to on-demand queries, meaning they persistently store data, and when a query is issued, they process it to retrieve a response. Hence, periodic and countinuous responses are not their normal behaviour. Therefore, if architects wish to support real-time continuous or on-demand queries queries, it often requires integrating these databases with other technologies capable of handling this type of query behavior (Section 5.1). However, in some cases, it's possible to leverage certain storage features to support continuous queries. For instance, some SQL databases like MySQL and Oracle use Materialized Views, Triggers and Stored Procedures, to obtain a behaviour similar to that of continuous queries. Materialized views significantly enhance database performance when executing specific queries by storing query results in a dedicated table. Instead of reconstructing the original query from the source tables each time, the database engine can efficiently retrieve the relevant data directly from the materialized view table[33]. Stored Procedures, on the other hand, are predefined queries within the database that can be triggered in response to specific events, allowing real-time data analysis and modification. Triggers are designed to automatically execute actions based on events that occur in a database[34]. By integrating these three components, it becomes possible to achieve behavior similar to that of a continuous query. This is accomplished by defining a stored procedure, triggering it with an event, and storing the resulting data in a materialized view. Furthermore, both column-based and document-based databases allow the creation of materialized views that aggregate and analyze data in real-time. These materialized views can be used to execute continuous queries on real-time data. Some databases also offer the ability to use User-Defined Functions (UDF) and triggers. For example, Cassandra allows the definition of UDF and triggers that can be used to perform specific actions in response to real-time events.

Language We have defined in Section 3.6 the language features as the capability of the components to provide some language to manage and define queries over data. After reviewing the official documentation of various technologies, it can be observed that database systems, both SQL and noSQL, provide the ability to interact with them with a declarative language as first ability. Then, some of them can also allow for imperative/OOP and/or functional implementations to enhance the expressiveness of the language. The SQL language, used by SQL database systems, is a declarative language that allow to incorporate functional, imperative, and object-oriented elements through procedures, scripts, and external frameworks. For example, it supports aggregating functions such as `SUM` or `AVG`. Moreover, using an ORM (Object-Relational Mapping) such as Hibernate [8], it is possible to handle objects in an object-oriented manner while interacting with

the database. For NoSQL databases, the query language varies significantly depending on the specific type of database. Document-based databases provide a primarily declarative language in which users use predefined functions like `find` to specify what they want as the search result. These languages can offer functional paradigms for aggregation or mapping functions. Additionally, they may also provide the ability to integrate user-defined functions with imperative and/or object-oriented programming languages. For instance, in MongoDB, you can use the find function to query documents in a collection and specify the criteria for the search, such as filtering by specific fields or values. Additionally, you can use functions like `mapReduce` following a functional paradigm, or follow imperative and/or iterative approaches to specify new functions. MongoDB language supports the use of JavaScript to define User-Defined Functions (UDFs): one can write the logic using JavaScript, which is an imperative language and can incorporate OOP concepts. However, notice that OOP paradigms are not used in this context. Column-based databases primarily offer a declarative language similar to SQL, in which users define declarative queries to retrieve data based on columns and conditions needed. Additionally, they support the use of imperative paradigms for operations involving more complex data processing logic, which can also be defined using functional paradigms. For example, the Cassandra Query Language (CQL) is a language very similar to SQL that supports declarative queries through SELECT, imperative operations like UPDATE, and can also follow functional paradigms, such as aggregation functions like AVG. Additionally, it allows the definition of user-defined functions using imperative paradigms. Key-value database languages are primarily imperative languages oriented toward data insertion and retrieval based on a key and usually do not support complex query languages. As a result, support for functional paradigms is mainly limited to atomic operations with a simple logic (e.g., additions or increments) even though in most cases, these functions are managed externally to the database. Declarative paradigms can be utilized within the database, although even in these cases, the typical approach is to define them within applications that adhere to the declarative paradigm. In such cases, it's the application code that manages declarative logic, while the database primarily offers fundamental key-value operations. For example, the Redis CLI language supports operations like `SET` and `GET` and allows for the definition of complex logic using scripts in the LUA programming language (which can be used both with a declarative and imperative approaches), even though the common approach is to handle complex logic outside the database. Functional commands supported by Redis CLI Language are primarily used for atomic operations such as increments (`INCR`) or additions (`ZADD`), and for executing Lua scripts (`EVAL`). Graph databases are characterized by higher complexity and offer specialized languages for managing and querying graphs. These languages enable both declarative and imperative approaches: declarative

queries can be used to search for data within the graph, specifying patterns based on relationships between nodes and edges to search for, and simultaneously imperative logic for more complex modification operations can be employed. Functional paradigms are not commonly used in these contexts, although they may be supported in some cases. For example, Neo4j provides the Cypher language, which allows you to search for patterns within the graph using the `MATCH` clause. Additionally, with Cypher, you can define imperative operations, such as adding a node or creating a relationship between nodes using the `CREATE` clause.

Extensions We have defined in Section 3.7 *Extensions* as the ability of a component to adeptly manage and process data in alignment with their specific characteristics. We have defined *Temporal* as the capability to account for data within their temporal dimension, providing functions for managing this data effectively, and Geospatial as the capability to analyze data within their spatial dimension, offering functions to extract information based on geographic locations. All databases can be organized to handle temporal and spatial data in their native implementations, adapting the data structure to the specific architectural requirements. SQL databases allow the use of temporal data types to define their tables (e.g., `DATETIME`) or to apply functions for modifying, filtering, or aggregating temporal data (e.g., `DATEADD` or `DATEDIFF`). Additionally, some SQL databases offer specialized extensions to enhance performance in handling this type of data. For instance, TimescaleDB is an extension of PostgreSQL that provides specific functionality to optimize performance and efficiency in managing and querying large volumes of temporal data[17]. The same applies to geographical data. SQL databases enable the use of geographical data types to define tables and provide functions for calculating distances or comparing geographical areas, such as `POINT`, `ST_DISTANCE` and `ST_CONTAINS` in MySQL. Furthermore, some SQL databases offer dedicated extensions for geographical data. For example, PostgreSQL provides the PostGIS extension[13]. NoSQL databases allow for the adaptation of data structures to include temporal information. Document-based databases enable the addition of temporal fields to documents and subsequently querying data based on them. Similarly, column-based databases allow for the inclusion of temporal columns in tables. Graph-based databases allow for the addition of nodes and relationships representing temporal events, while key-value databases can incorporate temporal data by adding a timestamp to key-value pairs. In general, column-based and graph-based databases are most suitable for this type of analysis. Columnar databases like Apache Cassandra or Apache HBase are ideal for temporal data that requires rapid access to data based on specific columns, especially for storing time series data where reading specific columns is common[43]. Graph databases like Neo4j are effective for addressing complex

questions about temporal connections and relationships. Document-based databases may be suitable when there is a need to store temporal information about data but complex analyses with fast access to columns are not required. Additionally, there are NoSQL databases specifically designed to handle temporal data with high performance. For instance, ClickHouse is an open-source columnar database optimized for high-performance analysis, particularly well-suited for managing temporal data and time series[53]. Thanks to its columnar structure, ClickHouse allows for efficient execution of complex analytical queries on large volumes of temporal data. Regarding geographical data, different types of NoSQL databases allow for adaptation in various ways. Document-based databases provide specific data types for incorporating geographical attributes into documents (for example, MongoDB and Couchbase offer the GeoJSON type) and offer geospatial query capabilities. Graph-based databases are particularly well-suited for this type of data, as locations and paths between locations can be represented as nodes and edges in a graph, enabling advanced geospatial queries and analyses. Column-based databases are typically not the optimal choice for geographical data, but it is still possible to design the schema by including columns for latitude and longitude to perform spatial queries. Additionally, there are extensions that provide tools for storing and analyzing geospatial data in NoSQL databases. For example, GeoMesa[6], which provides spatio-temporal data persistence on top of distributed databases for massive storage of point, line, and polygon data. For example, it can be integrated with various databases such as Cassandra or HBase and enables large-scale geospatial querying and analytics on distributed computing systems. Also, RSDB, which rely on HBase, can be used for remote sensing, which allows to efficiently monitor the physical properties of the environment using multi-temporal data acquisitions of large area[59].

Parallelism We have defined in Section 3.8*Parallelism* as the capability of the component to distribute data processing and/or storage across multiple nodes to improve performance. We have define *Data* Parallelism as the ability of the component to split the data-set to parallelize the operations, while *Task* Parallelism as the ability to split the query in tasks to be applied in parallel. Both task and data parallelism abilities can be offered by both SQL and NoSQL databases, although they may differ in the manner in which they provide them. In general, it can be stated that many databases, both SQL and NoSQL, may implement a combination of data parallelism and task parallelism techniques to enhance performance during executions whenever possible. However, their implementation in a specific situation depends on various factors. Firstly, actions undertaken in parallel must be independent; otherwise, there is a risk of data consistency issues and operation conflicts. For instance, if two concurrent operations attempt to

modify the same data portion without proper management, a race condition or data conflict may occur. Additionally, the use of imperative or Object-Oriented Programming (OOP) paradigms might restrict parallelism possibilities, as they often involve a sequential flow of control and operations dependent on prior outcomes. Therefore, to fully leverage parallelism, programming paradigms better suited for parallel computing, such as the functional paradigm, are preferred. Also, the structure of the database and the architecture of the data management system can influence the ability to implement parallelism. Some databases are specifically designed to support parallelism, while others are not. SQL databases can support parallelism, but their ability to fully harness it may vary. In SQL databases, queries can run concurrently across multiple threads or processes. SQL databases such as Oracle, SQL Server, and PostgreSQL support query parallelism. This means that a complex single query can be subdivided into sub-queries and executed simultaneously on multiple computing units. Moreover, in SQL databases, data partitioning, which involves dividing data into smaller segments known as partitions, can be implemented. These partitions can be distributed across separate servers, enabling parallel query execution on these segmented data sets. For example, PostgreSQL provides support for data partitioning. About NoSQL databases, column-based databases like Apache Cassandra are typically designed to provide a high level of parallelism in read and write operations. By distributing data across multiple nodes, they enable efficient parallel queries on specific columns. Document-based databases like MongoDB can support parallelism by distributing documents across separate nodes and allowing parallel execution of queries across different nodes. Key-value databases like Redis are primarily designed for read and write operations on individual keys or values and are not oriented towards parallelism. Graph databases like Neo4j are designed to manage data with complex relationships. They can efficiently execute graph queries, often involving significant parallel processing during graph traversal and relationship analysis. The approaches used to partition the data-set and execute queries in parallel on each portion of the data-set vary between SQL and NoSQL. Therefore, while for the purpose of our model it can be asserted that both SQL and NoSQL solutions can enable data parallelism when feasible, it's important to recognize that they implement distinct techniques. SQL databases use a partitioning technique, while most NoSQL databases employ a sharding technique. The main distinction between sharding and partitioning is that in the case of sharding, data shards are distributed across different servers or separate server clusters[47], whereas in the case of partitioning, data partitions may reside on different nodes within the same server or on different nodes within the same cluster. Thus, sharding involves distributing data across separate servers, whereas partitioning can involve dividing data across different resources within the same infrastructure. This is one of the reasons why NoSQL

solutions are generally more scalable [28].

Processing Actions We have defined in Section 3.5 *Processing Actions* as the capability of the component to provide actions to process the data and extract derived information. After reviewing the official documentations, we can state that most databases provide the capability to perform processing actions. The SQL language used with SQL databases offers clauses to allow for all processing actions. It allows for filtering using the `WHERE` clause, aggregation using functions like `SUM` and `AVG`, or aggregation based on specific attributes using `GROUP BY`. Additionally, SQL enables table joins through the `JOIN` clause. To perform pattern matching, the `LIKE` clause followed by a regular expression pattern is utilized. Processing actions capabilities are provided by NoSQL databases in different ways`moniruzzaman2013nosql`. Column-based databases enable data filtering, aggregations, grouping on one or more columns, and joining on tables or columns. Some of them also allow for pattern matching operations using regular expressions or search functions. Key-value databases offer limited support for these operations. Filtering is based solely on the primary key, and only a few of them allow pattern matching on patterns within values associated with the primary key. Aggregations, grouping, and joins require application-level handling, as key-value databases do not provide native support for these operations. Graph-based databases support these operations to be performed on nodes or edges. Of particular importance is pattern matching, which is fundamental for these databases as it allows searching for specific patterns on nodes and edges within the graph. Document-based databases allow filtering based on specific document conditions, aggregating with functions like sum, average, etc., and calculating statistics on document changes. They also support grouping based on specific document fields. Some of them even support joining between documents based on defined conditions and pattern matching operations, for example, searching for common values in document fields.

Persisted Data Type We have defined in Section 4.1 *Persisted Data Type* feature as the capability of a component's storage to persist specific types of data, which can be structured, unstructured or semi-structured. As previously mentioned in the introduction to this chapter, different database systems possess the capability to persist various types of data. SQL databases, based on the traditional structured query language, are primarily designed for managing structured data. These databases excel in organizing and querying data that follows a well-defined schema, such as that presented in tables with clearly defined columns. On the other hand, NoSQL databases have been specifically conceived to address a broader range of data. These systems stand out for their ability to handle not only structured data but also semistructured and unstructured data effectively. This

flexibility makes them suitable for applications where data structure may be variable or even absent.

Isolation Relational databases implement Concurrency Control mechanisms as a fundamental part of ensuring the transactional ACID properties[43]. These properties are a primary motivation for choosing relational databases. Looking at the Oracle documentation, we can see that it relies on Lock-Based mechanisms to isolate transactions during their execution. Oracle allows the programmer to select the most suitable level of locking between row-level, table-level and database locking. Moreover, Oracle's default isolation level is read committed, but it allows, with an appropriate command, to set the serializable isolation level. Looking at the PostgreSQL documentation, we can read that it implements a MVCC mechanism also for serializable snapshot isolation level. Its standard is read committed, but it can also provide repeatible read and serializable isolation level. Also, PostgreSQL provides table-level, row-level and page-level locking to be set up by the designer when MVCC is not enough. They are primarily employed in SQL databases to ensure ACID properties. As a consequence, they are are not commonly used in NoSQL databases, since to handle numerous concurrent read and write requests, some of them relax concurrency restrictions[35]. Graph databases typically employ Multi-Version Concurrency Control (MVCC) mechanisms because graphs can be intricately connected, and different versions help avoid conflicts. Columnar databases also usually employ MVCC strategies, both to track versions for historical analysis purposes and to handle conflicts. Systems like HBase, Hypertable, Bigdata, and GraphDB utilize the storage of different versions not only for conflict resolution but also for versioning purposes.[35] . CouchDB implements MVCC too. Similarly, Key-Value databases provide high concurrency, and the most common strategy is Timestamp-Based, where a timestamp is associated with each key-value pair[45]. However, this approach can be costly, as it demands significant storage capacity, a garbage collector to remove unused versions, and an algorithm to resolve conflicts in case of inconsistency. As a result, some NoSQL databases employ optimistic locking techniques, such as Neo4J. Before committing changed data, each transaction checks whether other transactions have made conflicting modifications to the same datasets. If conflicts arise, the transaction is rolled back. Applications that don't encounter many conflicting operations can opt for optimistic locking, supported by databases like Voldemort, Redis, and Membase[35] In general,

Atomicity To ensure ACID properties all relational databases implement some Commit Protocol mechanisms. They usually implement 2PC, such as Oracle, PostgreSQL or MySQL. Similarly to the 2PL protocol, the 2PC protocol is not commonly utilized in

NoSQL databases, as it is more typical of SQL databases for transaction management. Some NoSQL databases may provide features that helps in the implementation of the Sagas pattern to maintain data consistency in distributed transactions. For in- stance, Cassandra and MongoDB allow the implementation of compensating actions using the transaction log. Alternatively, in other cases when multiversioning is implemented in the database (e.g., Cassandra), it can be leveraged to facilitate the implementation of compensatory actions. However, it's important to note that using Sagas requires customization of application logic and is not a native feature of the database. Similarly, the deterministic approach can also be implemented at the application level. To simplify the implementation of this approach, it is necessary for the NoSQL database to support timestamp management and conflict resolution.

Availability All relational databases offers Replication, each with its own strategies. Looking at the documentations of various relational databases, it is possible to understand the different replication options they offer. For example, MySQL replication is asynchronous by default: the source writes the changes to a binary log, and replicas retrieve them when they want, without the need to be always connected to the net- work. Moreover, MySQL allows architects to select what to replicate, and to decide the scope of replication. In fact, replication can be used for backups, either by using the mysqldump tool[11], which is inefficient for large databases,or by putting the server in a read-only state. Moreover, Replication can be used for improve performance during the execution of queries mostly composed of reads, by distributing reads across replicas. In addition to the standard asynchronous replication, MySQL allows you to implement semi-synchronous replication.

The majority of NoSQL databases implement Replication. For instance, Cassandra utilizes a leaderless mechanism through a Gossip protocol designed to meet scalability and resilience requirements. Couchbase employs a Leader-Based solution with Automatic Leader Election, dynamically selecting leader nodes. Regarding data propagation, NoSQL databases generally implement it asynchronously to enhance write latency. This happens because NoSQL databases primarily focus on providing high availability, at the expense of data consistency[29]. As mentioned in the introduction, NoSQL databases are designed to scale horizontally and employ techniques like partitioning and distributed replication across multiple nodes. In accordance with the aforementioned CAP theorem, NoSQL databases maintain partition tolerance (due to horizontal scaling through partitioning) and prioritize high availability, accepting a certain level of compromise on data consistency[52].

Scalability Relational databases can allows to partition large tables, either horizontally and/or vertically. In the first case, different rows of a table are associated to different partitions, whereas in the second case, different columns are associated to different partitions. As it happens for the other features, different relational databases allows different strategies for Partitioning. By looking at the PostgreSQL documentations, we can read that by defining a table as partitioned table it is possible to select which tables of a schema have to be partitioned. It allows both Key-Value and Hash Partitioning. Key-Value Partitioning can be either designed by range or by list: range partitioning allows the designer to select the ranges of keys for each partition, whereas list partitioning allows to exactly indicate the keys to be inserted in each partition. Notice that the key-value approach does not necessarily involve the primary key: PostgreSQL allows one to define which attribute of the table to consider when partitioning it, allowing the designed to partition also attributes for secondary indexes. Also, it allows to combine the partitioning strategies to create multi-level partitioning. By looking at the Oracle documentation, we can see that it offers the same partitioning strategies of PostgreSQL, confirming what is written in the introduction of this chapter: many databases offers the same strategies. NoSQL databases implement Partitioning as they are designed for horizontal scalability. Key-range partitioning is commonly used by Columnar and Key-Value databases, such as Cassandra and Redis. Document databases, on the other hand, typically employ Hash-Based solutions. For example, MongoDB uses a hash function over the primary key to distribute documents across different partitions. Graph databases have their specific technique based on graph partitioning. In this case, graph nodes are partitioned in a way that connected nodes reside in the same partition.

Durability Most SQL databases implement both of these methods, which are used for database state recovery purposes. The common approach is to start from the last checkpoint and re-execute the transactions recorded in the log. Oracle, MySQL, and PostgreSQL use checkpointing to write data to disk, allowing manual configuration of checkpoint intervals. Additionally, they enable transaction logging by default. However, even though it is highly discouraged as it would limit the database's recovery capabilities, it can be restricted or disabled by the database administrator. NoSQL databases uses replications also for durability purposes. For example, Amazon DynamoDB replicates data across at least three data centers, thus providing high availability and durability even under complex failure scenarios[45].

Replication in NoSQL databases not only enhances availability and durability but also provides the flexibility to replace failing nodes with other servers when needed[35]. In fact, NoSQL databases operate under the assumption that complete consistency enforcement

may not always be necessary in real-world scenarios. They prioritize achieving higher availability and partition tolerance even if it means compromising on consistency in adjustable ways. This trade-off can impact durability; if a system experiences consistency issues, it may jeopardize its overall durability, potentially leading to data loss[40]. To deal with recovery in case of failures, many NoSQL databases implements both of Checkpointing and Logging, and to start from the last checkpoint, followed by the re-execution of transactions recorded in the log. For instance, in the case of the in-memory Key-Value database Redis, data is asynchronously saved to disk and provides the option to configure periodic checkpoints. For example, MongoDB employs an append-only log for write operations, which is used also for monitoring purposes. The columnar database Cassandra, on the other hand, relies on a write-ahead log and periodically generates snapshots of SSTables. In the context of Graph databases, Neo4j utilizes a transaction log.

6 | Methodology Overview

This chapter describes the software methodology to create and a data-intensive architecture and identify systems to implement the corresponding infrastructure. T he methodology assumes that the architect

- knows how many producers and consumers are present, what they produce and the queries they ingest in the system;
- has already carried out the requirement collection process, from which he/she has extracted a list of requirements to be covered, both functional and non-functional, when designing the architecture.

This information is used by the architect to create the methodology input in a format described in Chapter 7. This input is used in the methodology algorithm, described in chapter 8 that produces as output the set of concrete components to implement the architecture. The methodology involves the use of the model shown in Figure 6.1, which has been described in Chapter 2, Chapter 3 and Chapter 4: starting from this model, and following the first part of the methodology, components and features are activated, and the *logical data-intensive architecture* is obtained. In the second part, the methodology guides the architect to select the set of physical components necessary to implement the architecture. To do this, the architect must follow what is indicated in the Chapter 5. The final output of the methodology is the set of systems necessary to obtain the physical architecture that corresponds to the logical architecture obtained in the previous step.

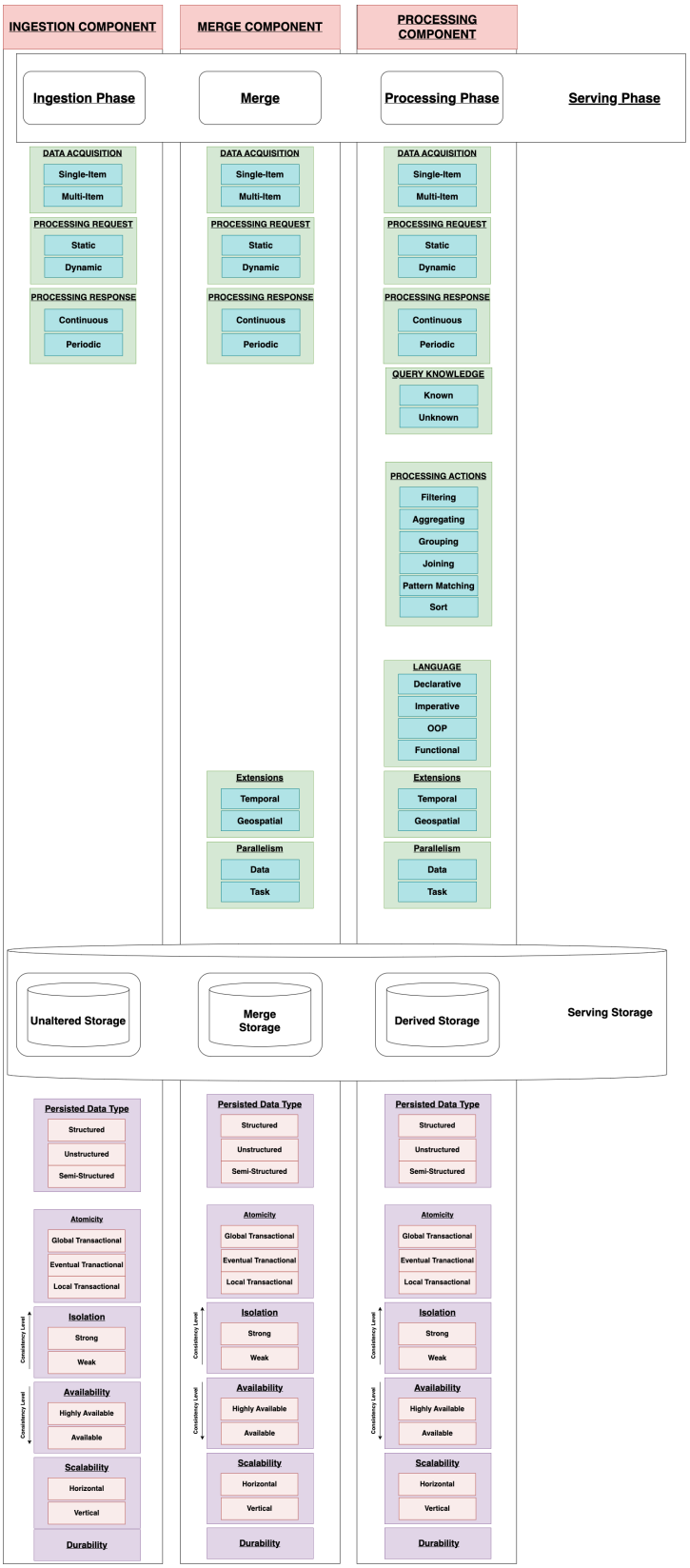


Figure 6.1: Complete model.

In the following it is shown when to take each decision following the methodology, without specifying the criteria by which decisions are taken. These will be specified in chapter 8, where the full algorithm of the methodology is provided.

The methodology is a systematic procedure that takes as input the architecture, which is composed of multiple data-flows. This is modeled by the architect with a structure described in Chapter 7 considering the information previously collected. Each data flow corresponds to a type of query that can be performed in the system that, as it will be shown, can be decomposed in sub-queries: a sub-query is a unitary computation action that can be performed over the data; this action has its own characteristic both in terms of produced data and of requirements. A sequence of sub-queries determines the final query, which correspond to a data flow. As a consequence, each data flow is associated with a query, composed of sequence of sub queries, each with its own characteristics.

Figure 6.2 shows the pseudo-code of an overview of the methodology. Then, Chapter 8 will show a more detailed pseudo-code.

```
main (infrastructure) {
    // identify data flows
    dataflows = find_subgraphs(infrastructure);

    // parametrize data flows
    dataflows = parametrize(dataflows);

    // STEP1: From Data-Flows to Model Mapping Instances
    model_instances = methodology1(dataflows)

    // STEP2: From Model Mapping Instances to Architecture Mapping
    architecture_model = methodology2(model_instances);

    // STEP 3 : From Architecture Mapping to Architecture Systems
    architecture_systems = methodolgy3(architecture_model);

    return architecture_systems;

}
```

Figure 6.2: Methodology pseudo-code.

Starting from the information that the architect is supposed to have gathered, the methodology guides to identify a set of technologies for each data flow. To do that, for each data flow the architect has to first create an instance of the model, and then to activate components and features according to the input's characteristics.

Remembering that

- The *Ingestion* component is always present;
- The *Serving* component is an extension of the other components to allow in-memory processing and/or management;
- The *Merge* component is required when data coming from diverse components has to be unified before being used;
- The *Processing* component is required when queries require some computation to produce the final output;
- The *Unaltered* storage is required when the data to be persisted is the one that comes from the outside as it is;
- The *Merge* storage is required when data needs to be unified before being stored;
- The *Derived* storage is required when derived information needs to be persisted.

In the following, we refer as a component to be instantiated to say that it is activated and their abilities selected. An *Ingestion* component is associated and instantiated for each producer to ingest data coming from them. Moreover, if necessary, the storage is instantiated and optionally in-memory activated. The *Merge* component is instantiated any time data needs to be unified. Then, by considering the query associated to the data flow, the methodology guides the architect to decide whether to activate instances of the *Processing* component (and related storages) or not for each sub-query. Considering this, an instance of the model can have more than a *Ingestion*, *Merge* and *Processing* component, and each component may or may not activate its storage to persist the result of that specific computation. When an ability corresponding to a feature is activated we say that the system we have to chose must provide that ability. After completing all the steps for each data flow to establish a mapping with our model, the architect must consolidate the various instances, resulting in the final architecture aligned with the model. In this phase, components may be consolidated when they are tasked with delivering identical capabilities or disaggregated when they are featured in different data-flows, each requiring mutually incompatible capabilities. The next step is to select the appropriate set of systems to implement the architecture. Chapter 5 shows how to perform the *mapping*: systems useful to implement *State-Centric* components mostly cover *Storage Features*, but may be able to provide also *Computation Features*, whereas those useful to implement *Data-Centric* components cover *Computation Features*. So, looking at the mapped model, for each phase, the activation of the storage highlight the need for a database technology, whereas the absence of the storage highlight the need for a processing technology.

7 | Methodology: Input Requirements

In this chapter, we will demonstrate the process of generating input for our methodology, starting from the architect's knowledge. The input for the methodology is derived from the information previously gathered by the architect regarding producers, consumers, queries, and requirements. Specifically, the architect needs to be aware of:

- How and where data is produced by producers.
- The queries that consumers use to request information.
- When and how consumers access the data.
- The requirements that the architecture must fulfill, both functionally and non-functionally.

Based on this information, the architect can create the necessary input for our methodology, which will be utilized in applying the algorithm described in Chapter 8.

To generate this input, the architect must define the infrastructure as a graph in which nodes represent the operations required to respond to consumer queries. Once the infrastructure graph has been defined, data flows that constitute the input of the methodology can be identified.

Furthermore, to effectively apply the methodology, the architect must also specify preferences for both functional and non-functional requirements, which we refer to as *criteria*.

Throughout the chapter, we first provide criteria for both computation and storage components in Section 7.1. Then, we explain how to model an infrastructure (Section 7.2), how to identify data flows based on the infrastructure (Section 7.3), and how to identify relevant parameters for the subsequent phases of the methodology within the nodes of the data flows (Section 7.3).

7.1. Decision-Making Criteria

In this section, we list a set of decision criteria that architects can choose from based on their requirements. By decision criteria, we mean the options architects can select, considering both the resources at their disposal and how they intend to manage the trade-offs that may arise in the requirements. These criteria will influence the choices that the methodology leads to make.

We have identified a set of criteria based on the literature review conducted for Chapter 5, after pinpointing the key differences among various systems. However, this methodology is flexible enough to accommodate the introduction of new criteria with minor adjustments to the systematic procedure. In the following sections, we will demonstrate how these criteria influence decision-making and indicate where modifications would be necessary when adding new criteria.

We start by enumerating the decision criteria pertaining to the computation phase (Subsection 7.1.1), followed by the enumeration of those concerning storage (Section 7.1.2).

7.1.1. Computation Criteria

Regarding the computational phase, we have identified three main criteria to consider for our choice. These criteria are related to the trade-off between latency and operational cost: to achieve low latencies, it is often necessary to use technologies that are more resource-intensive and complex to manage. On the other hand, to reduce operational costs, it may be necessary to accept higher latency.

CC1: Cost Reduction Criteria We define *Cost Reduction Criterion* as the approach that prioritizes minimizing resource expenditure at the expense of response latency. Under this criterion, low-latency solutions are chosen only when strictly necessary and explicitly requested by a consumer. Hence, cost-effective solutions are favored, even at the expense of latency, but pre-processing solutions are implemented whenever possible to anticipate the final computation. This way, when computation is requested and needs to begin, a portion of it has already been completed, thereby partially reducing the workload and its impact on latency.

CC2: Latency Reduction Criteria We define *Latency Reduction Criterion* as the approach that aims to minimize latency as much as possible, even if it results in increased processing costs, even when not explicitly requested by the consumer. Under this criterion, low-latency solutions are employed in all cases unless there are exceptional circumstances

where it is not feasible.

7.1.2. Storage Criteria

In this section, we list the criteria among which the architect can choose when deciding to activate storage for a component. These criteria have been identified based on the information collected in Chapter 5, where we mapped state-centric systems to our model. We have identified the key factors that influence the choice of the type of state-centric system to implement. Based on the selected criteria, the methodology involves the activation of various skills related to storage features in the model. However, the choice of a criterion does not entail the activation of a skill for every storage feature. Features that are not directly considered by the criterion remain optional. However, given the presence of trade-offs that are managed by the selection of different systems, in cases where there is no technology capable of simultaneously covering both trade-offs, the methodology will choose the system that optimizes the trade-off most effectively, prioritizing requirements related to the selected criterion. We have identified two primary trade-offs that the architect can consider: the trade-off associated with meeting availability requirements, which may involve compromising on consistency (referred to as CS1), and the trade-off associated with scalability, which may entail resource costs (referred to as CS2). These choices are considered crucial when selecting the type of state-centric technology.

CS1: high Availability vs Consistency As discussed in Section 4.2, achieving high availability in distributed systems, where partitioning is employed, often requires sacrificing data consistency. Conversely, prioritizing strong consistency among replicas can lead to a trade-off involving the complete availability of the system[25]. Consequently, the architect can determine which of the two requirements to favor between `CS1 = high-availability` or `CS2 = strong-consistency` based on the specific needs of the application.mploying this criterion, the methodology assumes that the architect intends to anticipate the possibility of horizontal scalability, as it aligns with the CAP theorem by making the trade-off between these two aspects to ensure partition tolerance requirements.

CS2: ACID vs BASE The ACID properties are safety guarantees that can be provided by certain databases. ACID is an acronym for Atomicity, Consistency, Isolation, and Durability. Atomicity has been discussed in Section 4.3, isolation in Section 4.4, and durability in Section 4.5. In the context of ACID properties, Consistency takes on a different meaning compared to the CS1 criteria. Here, it refers to the preservation of all database rules, such as unique keys[25]. This ensures that the database remains in a

consistent state with respect to its invariants. BASE criteria stands instead for Basically available, soft state and eventual consistency[37]. The first principle, Basically Available, means that a system must remain essentially available to handle user requests, even if it may not be in a completely consistent or reliable state; soft state refers to the fact that data can be in a transitional or temporarily inconsistent state; eventual consistency highlights that in the long term, all data will converge to a consistent state. Hence, the architect that wants to include this requirement in its input has to insert either `CS2 = acid` or `CS2 = base`.

Please note that we have chosen to separate these two criteria because, although they may appear quite similar, they have significant differences in terms of the covered requirements. In fact, the concept of 'consistency' varies between the two criteria: in CC1, it refers to the consistency of a single copy of the data, while in CC2, it pertains to the integrity constraints defined by the database that must always be upheld across all nodes. Consequently, the definition in CC2 represents a stricter and broader requirement[25]. Furthermore, the consistency in CC2 is not strictly a property of the database but rather an application-level property: in the context of ACID, the application can leverage the properties of atomicity, isolation, and durability to achieve consistency[37].

We have also identified a third criterion, which we denote as `CS3`, where the architect is not interested in specifying any preference but only wants data to be persisted in some way.

7.2. Infrastructure Modeling

The architecture should be modeled as a Directed Acyclic Graph (DAG). A DAG consists of a finite set of edges and vertices, with each edge directed from one vertex to an- other. DAGs are very close to workflows, but they do not include cycles. [24] DAGs has been widley used in the topic of distributed processing systems to model the pipeline of actions to be performed to process the stream of data. In our methodology, architects use it to model the architecture corresponding to the infrastructure they want to design.

The components of the graph are depicted in Figure 7.1.

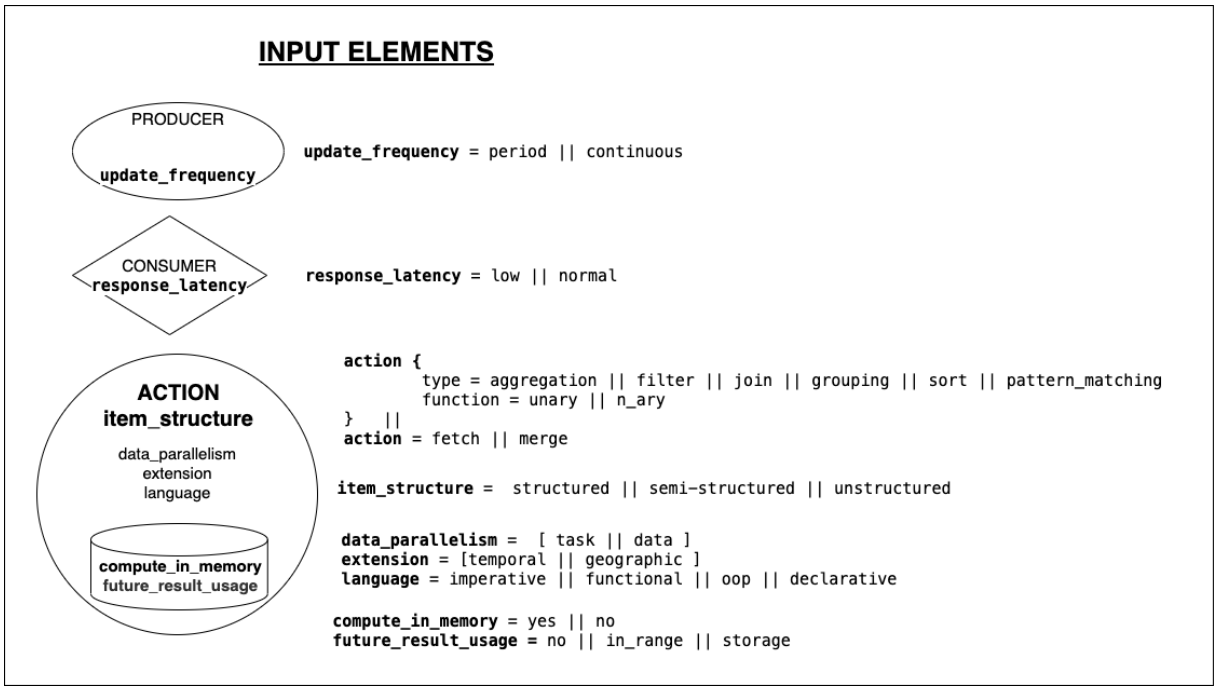


Figure 7.1: Elements Composing the Input

The input and output elements are, respectively, referred to as *producers* and *consumers*. According to the definitions we have provided, producers are those who input data into the infrastructure, while consumers are those who request data, which may be more or less processed. They are not under the control of the infrastructure, but the architect must know them and assign a unique identifier to each of them. Additionally, for each producer, the architect must specify the frequency at which they ingest data into the infrastructure, which we define as *update frequency*. This frequency can be categorized as *periodic*, if the producer sends new data at defined time intervals, or *continuous*, if it provides new data in a continuous flow. Regarding consumers, the architect must also specify the latency within which they expect to receive a response, which we define as *response latency*. This latency represents the period of time within which a consumer anticipates the processing and return of the requested data. We categorize this response as *low* when there is a critical need for minimal latency on the consumer's end, and as *normal* in other scenarios. The same consumer can consume data from different flows, and in such cases, they may have different latency requirements for each flow.

Internal nodes represent atomic actions in which queries are decomposed. For each of these actions, the architect has the option to specify whether they want the produced data to be retained for future use. To indicate this preference, the architect uses the *future_result_usage* parameter, which can be configured in three ways: *no*, *in_range*, or

storage. In the first case, there is no intention to save the data generated by the action for future use. In the second case, the architect anticipates using the data produced by the action within a specified time-frame. In the third case, there is a desire to permanently retain the result of the action.In this latter scenario, the architect specifies the criteria that have to be followed by the methodology among those highlighted in Section **??**.

The actions can be categorized into three main groups: *fetch* actions for data retrieval, from both original sources and other actions involving storage; *merge* actions used to integrate data from various sources; and *processing* actions, which represent the actual detailed processing actions as elaborated in Section **??**. Fetch actions involve only one producer and are limited to retrieving the data as it arrives. In the case of an external producer, this occurs at the ingestion rate, without any modifications. When a fetch is connected to a database, it represents a select operation within the database (i.e., as categorized based on our queries, a known query on demand). Merge actions are employed to amalgamate data from various sources in order to enable coherent processing by the subsequent action.

In the case of processing actions, the architect must also specify whether the function in question is *unary* (i.e., parametric on a single item) or *n-ary* (i.e., parametric on multiple items). Furthermore, the architect must specify the structure of the data passing through the node, which can be *structured*, *semi-structured*, or unstructured, as defined in Section 4.1. Lastly, for each action node, the architect can specify a set of optional features if they wish to anticipate specific functionalities. We define as optional features those features that are not relevant for the choice of the type of system to be selected, but they may influence the choice of the specific system. Considering the features of our model, we have identified data parallelism, extension, and language. In this case too, the methodology remains flexible, allowing for the inclusion of other optional features as determined by the architect.

Actions can have zero, one, or multiple outgoing edges. The merge action must have multiple outgoing edges, as it is used to combine and make data consistent from different sources. All other actions have only one incoming arrow.

In the upcoming figures, we will not reference optional features to maintain better visual clarity. However, it's worth noting that within each node, the architect can indicate a list of optional characteristics they desire for the corresponding action.

All known queries, both static and dynamic, are represented within the infrastructure. Additionally, the architecture allows for the anticipation of incoming queries that may not be known at the moment, using the *future data usage* attribute present in all actions. The

data flow, which illustrates how data enters, flows through, and exits the infrastructure, is represented by the edges connecting the various input elements. The basic scenarios for modeling the different relationships between nodes are depicted in Figure 7.2.

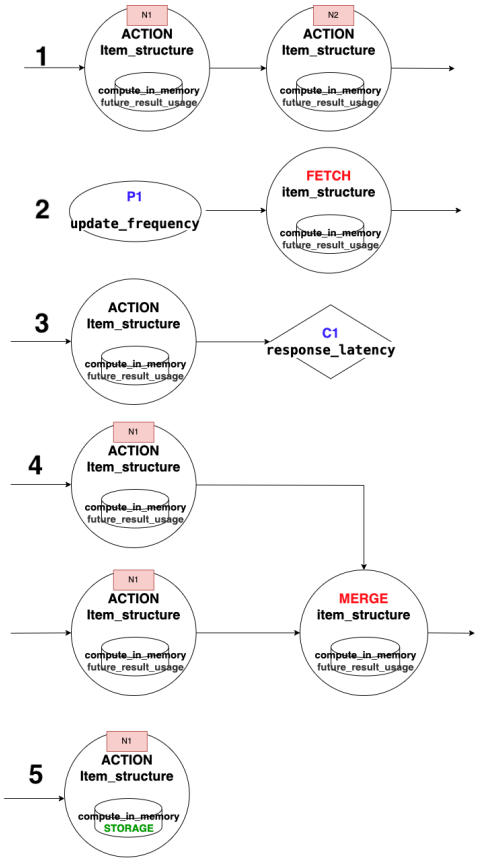


Figure 7.2: Nodes Relationships

.

The first example illustrates the connection between two generic actions. An edge connecting node1 to node2 indicates that node2 performs its action on data generated by node1. Actions can have one or more incoming edges and one or more outgoing edges. Producers, as in the second example, can only have outgoing edges since they introduce external data into the infrastructure. We consider them as external to the architect's control over the infrastructure and, for this reason, they must be connected to a *FETCH* action to retrieve the data they produce. Similarly, consumers can only have incoming edges. In this case, there are no restrictions on the types of actions to which they can be connected (as shown in example 3). The fourth example demonstrates how data integration from two different nodes occurs: to achieve this, a *MERGE* node must be inserted to combine the flow. In the last example, we show how to represent insertion operations in

the case of state-centric components. The last action in the flow has no outgoing edges, and the *future_result_usage* attribute is set to *STORAGE*. This represents a represents a flow entering the architecture for which we cannot predict when it will exit (i.e., when a consumer will request it). The same approach can be used for all other cases. For example, in the first example, if we place a *STORAGE* at N1, we indicate that, beyond the known query related to the flow connecting it to N2, we anticipate there may be future queries on the data produced by N1. By setting it to *IN_RANGE*, it is anticipated that this data will be requested within a specific time range, after which it may be discarded.

Infrastructure Modeling Example

To better illustrate the modeling process, let's consider a scenario of a simple sales infrastructure with physical locations and an online store. At this point, in addition to not considering optional parameters, we are also not taking storage into account. However, it's important to note that storage can be arbitrarily configured in each node at the architect's discretion.

Let's suppose we want to model an infrastructure for a sales company that manages two physical stores and an online store. The company has low-level managers in each physical store and high-level managers who oversee all the stores. Let's imagine that the low-level managers are interested in receiving a summary of products sold online and in their respective physical stores at the end of each day. On the other hand, the high-level managers want a summary of products sold online and in both physical stores, also at the end of each day. Furthermore, all managers want to be informed in real-time about online sales. Additionally, suppose that store B has technology that automatically records sales as soon as they occur, while in store A, sales must be manually entered. Online sales are naturally recorded in the system as soon as they occur. Finally, let's assume that high-level managers and store A do not have specific latency requirements for receiving daily information, while store B wants to receive it as quickly as possible.

Based on this information, we proceed to model our infrastructure. The first step is to identify producers, consumers, and queries. Additionally, we need to determine the update frequencies of the producers and the latency requirements of the consumers. It's important to remember that the same consumer can receive data from different flows and have different latency requirements for each of them.

Based on the information gathered, we identify three producers and their respective update frequencies:

Based on the information gathered, we identify three producers and their respective up-

date frequencies:

- Shop A (A), whose update frequency is periodic because sales are manually entered, and therefore will be entered from time to time, possibly in bulk.
- Shop B (B): whose update frequency is continuous because sales are automatically entered into the infrastructure as soon as they occur.
- Online Store (O): whose update frequency is continuous because, like B, sales are automatically entered into the infrastructure.

Furthermore, we identify four different queries, each corresponding to one of the four data flows in the infrastructure:

- *Query1A: sum per product of all products sold online and in Shop A daily.* This will be modeled with two "fetch" actions, one to retrieve data from source A and one to retrieve data from source O, followed by a "merge" action to combine the collected data. The "processing" actions include aggregation to calculate the sum per product (which will be a multi-item function) and sorting before delivering it to the final consumers (also multi-item).
- *Query1B: sum per product of all products sold online and in Shop B daily.* This will be modeled with two "fetch" actions, one to retrieve data from source B and one to retrieve data from source O, followed by a "merge" action to combine the collected data. The "processing" actions include aggregation to calculate the sum per product (which will be a multi-item function) and sorting before delivering it to the final consumers (also multi-item).
- *Query1M: sum per product of all products sold online in real-time.* This will be modeled with two "fetch" actions, one to retrieve data from source M and one to retrieve data from source O, followed by a "merge" action to combine the collected data. The "processing" actions include aggregation to calculate the sum per product (which will be a multi-item function) and sorting before delivering it to the final consumers (also multi-item).
- *Query2: sum per product of all products sold online, in Shop A, and in Shop B.* This will be modeled with three "fetch" actions, one to retrieve data from source A, one to retrieve data from source B, and one to retrieve data from source O, followed by a "merge" action to combine the collected data. The "processing" actions include aggregation to calculate the sum per product (which will be a multi-item function) and sorting before delivering it to the final consumers (also multi-item).

Finally, we identify three consumers, the queries they are interested in, and their latency requirements:

- Shop A Managers (mA): are interested in Query1A without low-latency requirements and in Query2 with low-latency requirements.
- Shop B Managers (mB): are interested in Query1B without low-latency requirements and in Query2 with low-latency requirements.
- High-Level Managers (mM): are interested in Query1M without low-latency requirements and in Query2 with low-latency requirements.

Based on the information gathered, we can represent the infrastructure as depicted in Figure 7.3. To enhance the figure's clarity, we have replicated the consumers to associate them with various query flows. Nevertheless, consumer nodes sharing the same ID pertain to the same physical consumer. Importantly, it should be noted that a single consumer may exhibit varying latency requirements contingent on the specific query flow.

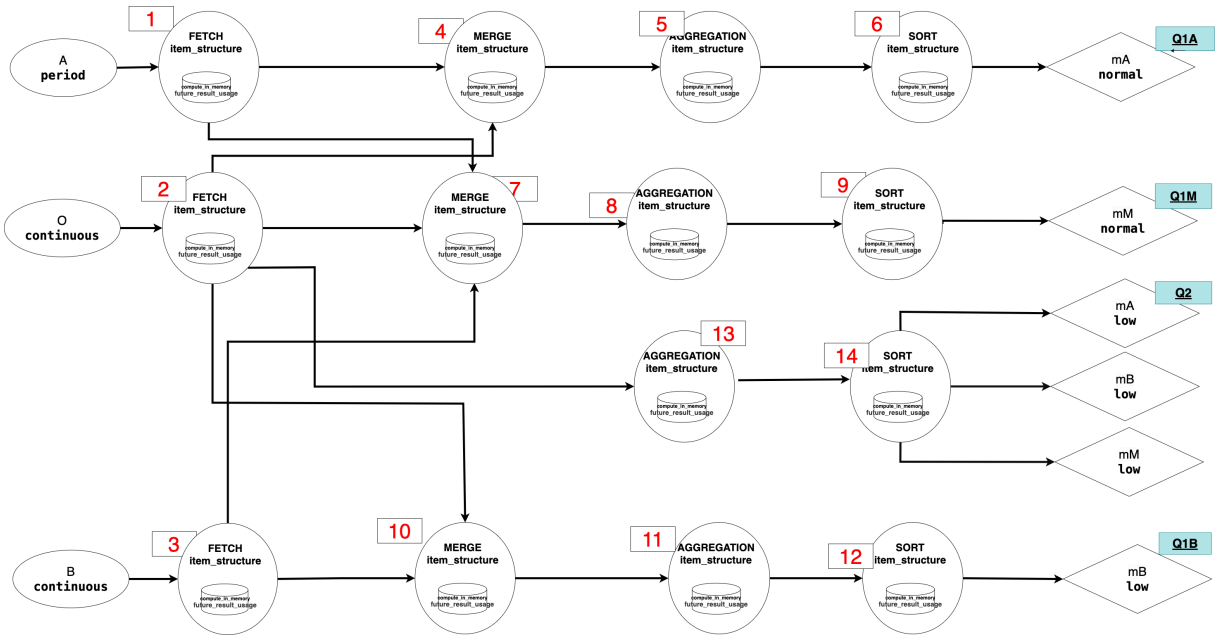


Figure 7.3: Nodes Relationships

. Note that the various nodes have been numbered. This numbering will be helpful in the next steps for identification.

From this point forward, we will systematically proceed to identify the various data flows, which constitute the input in the initial step of our methodology. The primary goal of this step is to achieve a mapping of the infrastructure with our model. Subsequently,

during the second phase of the methodology, we will utilize this mapping to identify a set of technologies suitable for our infrastructure.

7.3. Data-Flow Identification

The next step involves identifying the various data flows that data can follow within the infrastructure. To do this, any graph traversal algorithm can be implemented with the aim of identifying sub-graphs. By "subgraph," we mean a portion of the graph consisting of a set of nodes and the edges that connect them within the DAG. Nodes can be present in more than one subgraph. These different sub-graphs correspond to the various data flows within the infrastructure.

Continuation of the Example: Mapping Data Flows within the Infrastructure Graph

In the infrastructure described in the previous section's example, by traversing the graph, we can identify the sub-graphs corresponding to the data flows of the four queries. Figure 7.4. shows the data flows. It's worth noting that some components are present in multiple data flows, indicating that certain queries can share pre-computations.

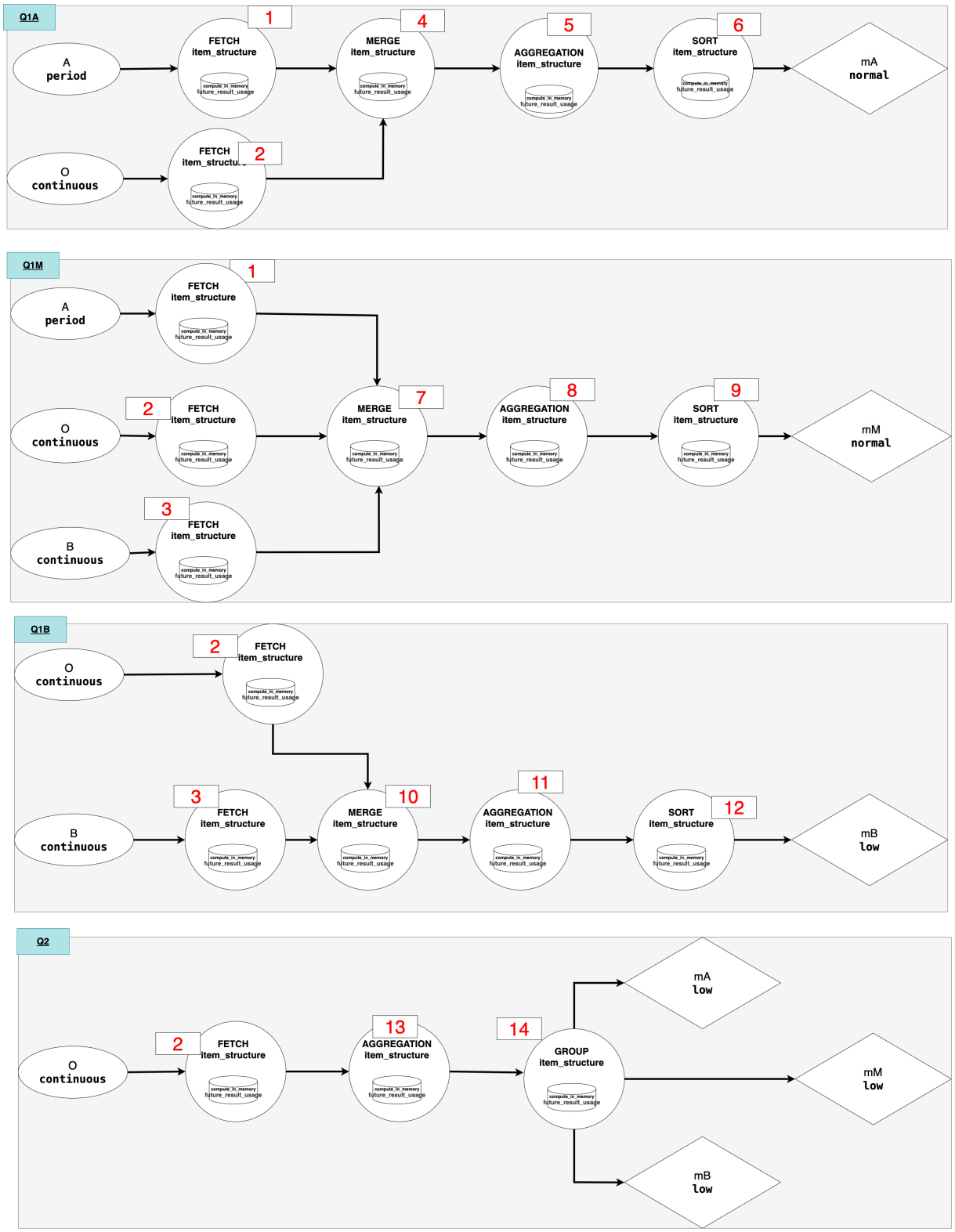


Figure 7.4: Nodes Relationships

.

7.4. Data-Flow Parametrization

Now we have identified the data-flows, and for each of them, we have determined the sequence of actions to be performed along with their respective properties. We have also identified the producers along with their update frequencies and the consumers with their response latency requirements.

For producers, we have defined the *update frequency* as the time interval between the ingestion of two pieces of data or data blocks. We have categorized this frequency as *periodic* when data is sent periodically, such as in groups of items, and as *continuous* when a continuous stream of items is transmitted. For consumers, we have defined *response latency*, which represents their latency requirements. We have categorized these requirements as *low latency* when the producer has stringent response latency requirements from the data flow and as *normal* otherwise.

Now, the methodology guides the architect to specify, for each internal node (i.e., action) of each data-flow, the required *update frequency* and *response latency*. The data-flows with this new information become the input for the first true step of the methodology, which is the mapping with our model. Figure 7.5 illustrates the elements that must comprise each data-flow. It does not include the attributes of optional features for improved readability. However, it is important to note that these attributes may be present in each action.

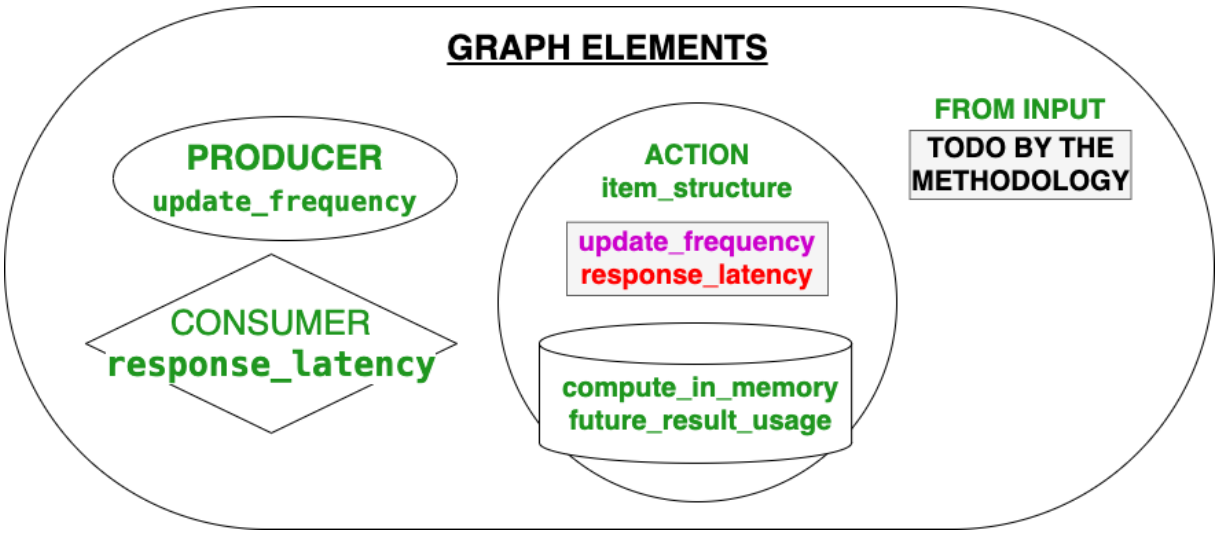


Figure 7.5: Nodes Relationships

.

We define the *update frequency* of an internal node as the rate at which the node must

request data from the preceding component, which can be *continuous* or *periodic*. Its *response latency* is defined as how quickly it must send the results of its action to the next component, which can be *low* or *normal*.

To set the parameters for each internal node, regardless of the criteria, we must traverse the graph starting from the consumers (on the right) and proceed towards the producers (on the left) to define the various response latencies. Subsequently, we proceed in the opposite direction to establish the update frequencies. The choice of these parameters depends not only on the producers and consumers but also on the processing criteria selected by the architect from those described in Subsection 7.1.1. We begin by importing the response latency from components directly connected to external consumers, considering the response latency required by those consumers themselves. It is important to note that in the case where a data-flow lacks a final consumer (such as insert queries), unless specific circumstances are specified by the architect in defining storage requirements, we consider the database to have a normal latency, as we can presume it won't be accessed immediately. In the case of CC1, the goal is to reduce costs unless reduced latencies are absolutely necessary. Therefore, if at least one consumer requires low latency, we assign a low response latency to the node. Otherwise, we set a normal response latency. This decision propagates backward until it reaches the producers. In the case of CC2, we aim to optimize latency at any cost. Consequently, we set a low latency in all nodes of the dataflow.

After defining all latencies, we start with the *FETCH* nodes directly connected to a source and traverse the graph in the opposite direction to define the frequencies of the intermediate nodes. The update frequency for each node is determined in the same way, regardless of the criteria. The update frequency of the *FETCH* node corresponds to the rate at which the producer connected to it sends data. For processing actions, the update frequency is propagated in the same way as that of the preceding node. To define the update frequency in *MERGE* operations, it is necessary to consider the update frequencies of all nodes being merged. If all nodes have the same update frequency, they are treated as a single flow, and their update frequency is propagated to the merge. If, on the other hand, the nodes entering the merge have different update frequencies, the flow is split into two, with one managed as continuous and the other as periodic. This strategy optimizes computations to be performed in advance. Figure 7.6 shows how to perform this split in the case of two nodes with different flows: actions performed after the merge are duplicated for the periodic and continuous flows, while the merge action is postponed to the end.

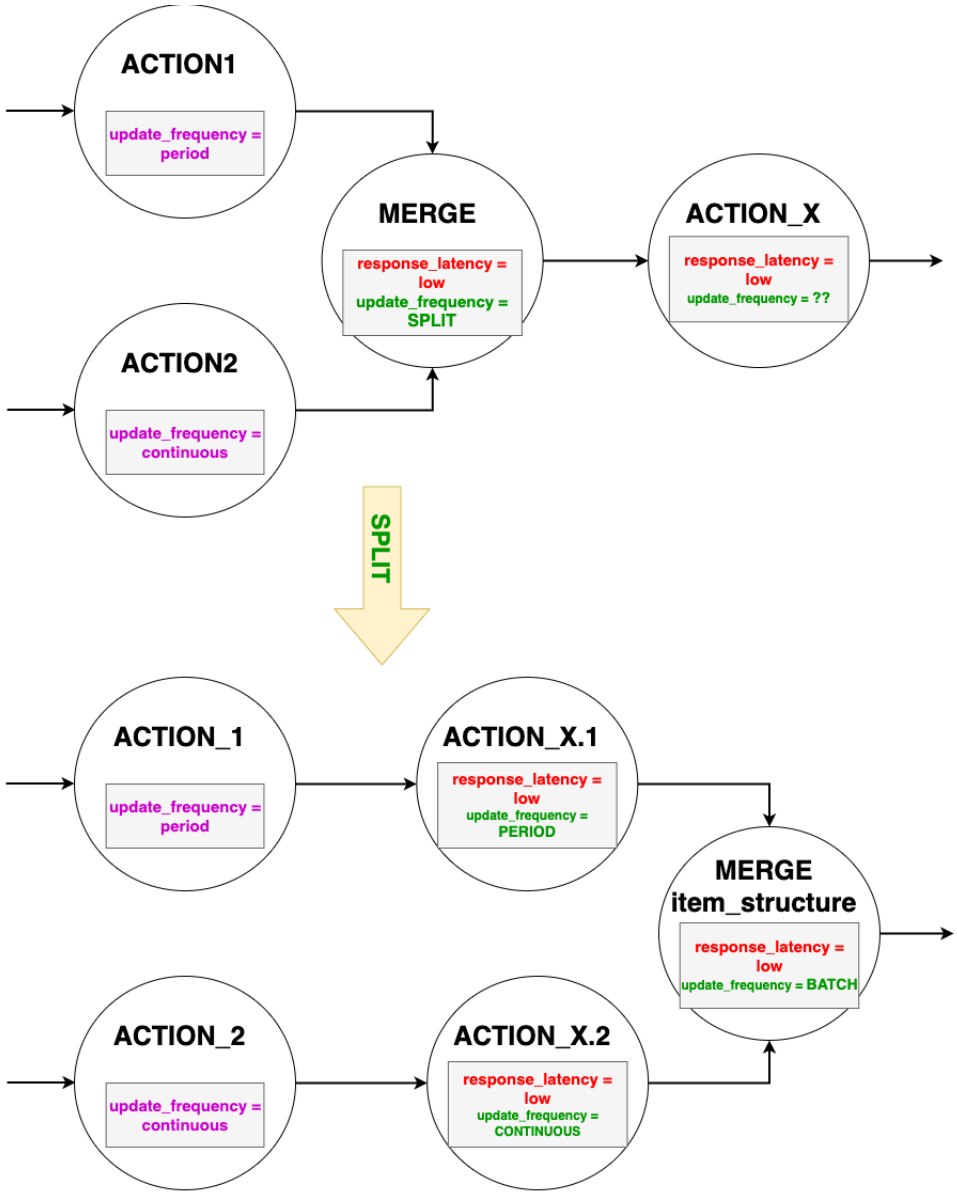


Figure 7.6: Merging two nodes with different frequencies

.

Figure 7.7 illustrates the procedure when there are more than two flows: in this case, the merge is divided into multiple parts to join nodes with the same update frequency.

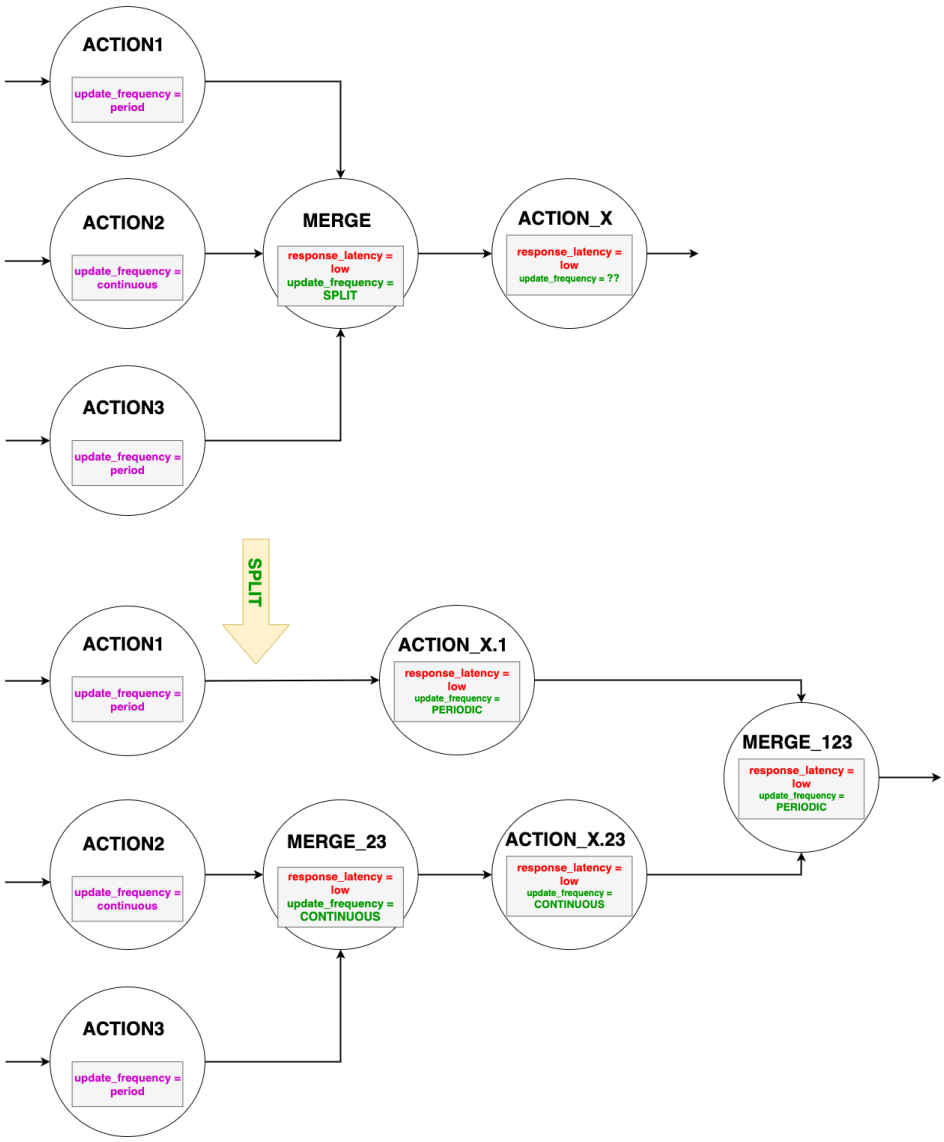


Figure 7.7: Merging more nodes with different frequencies

.

This step is customizable by the architect, who can tailor it by adjusting the computation criteria, thereby adapting the strategies for assigning internal node parameters.

Continuation of the Example: Assigning Parameters to each Data-Flows Actions

Building upon the previous sections' example, we have delineated the various data flows. To progress, we must pinpoint the intermediate parameters required to attain the output of the initial substantive step in the methodology. To showcase diverse applications, we

evaluate distinct query requirements and select parameters accordingly, aligning with the chosen computational criteria.

Query1A with CC1 requirement (Figure 7.8) Following the requirement CC1, we aim to prioritize latency only when strictly necessary. The consumer in question (mA) doesn't have stringent *response latency* requirements, so we uphold *normal* response latency. Regarding the *update frequency*, *FETCH* node N1 inherits a *periodic update frequency* from the corresponding source, while *FETCH* node N2 operates with a *continuous update frequency*. *MERGE* node N4 needs to merge two flows, one periodic and one continuous. To optimize this process, we decompose them into two flows, as explained earlier, and recombine them at the end. It's important to note that since the final operation is a sort, the *MERGE* will also arrange the partial results from the two flows as it combines them.

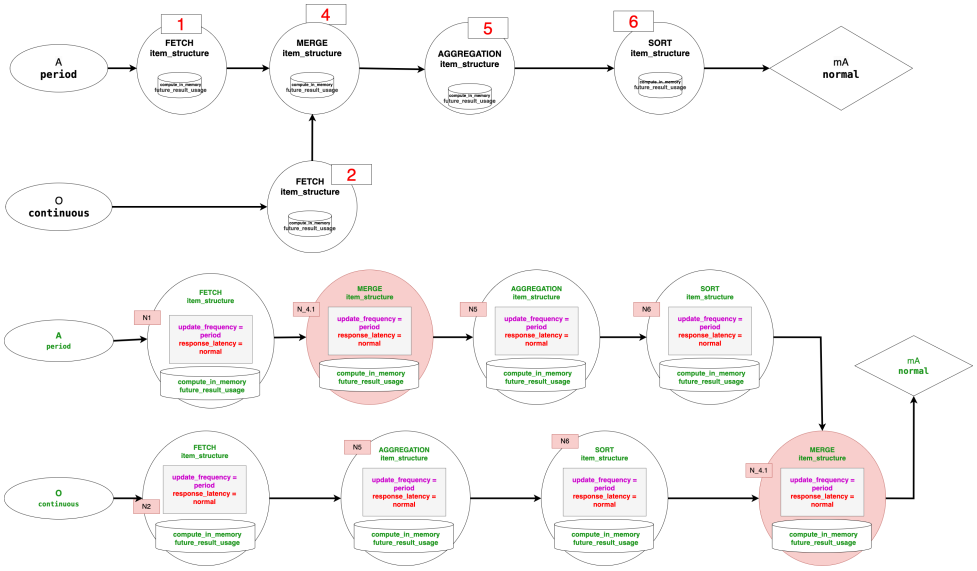


Figure 7.8: Query1A Parameters according to CC1

.

Query1B with CC2 requirement (Figure 7.9) For Query1B, we choose requirement CC2, which entails optimizing even when not explicitly requested by the consumer. In this case, the consumer has a *low response latency* requirement, so we must propagate this requirement to the nodes. However, even if it had been a *normal* requirement, we would still propagate *low* due to the requirement. Regarding the *update frequency*, *FETCH* nodes N2 and N3 inherit a *continuous* update frequency from their respective sources.

MERGE node N10 needs to merge two *continuous* flows, so it assumes a continuous update frequency, which also propagates to the other nodes.

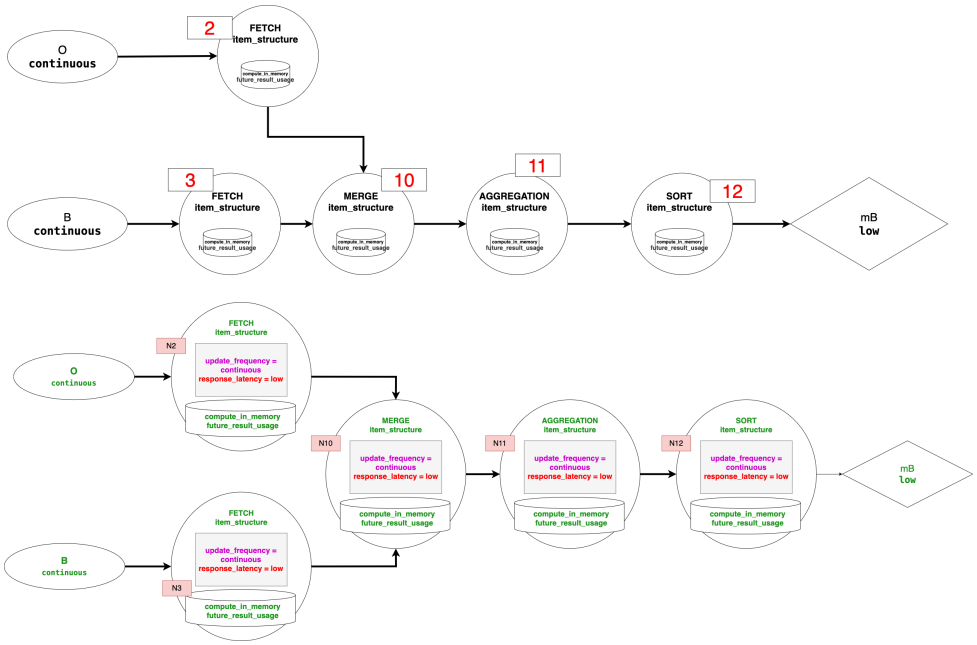


Figure 7.9: Query2A Parameters according to CC2

.

Query1M with CC1 requirement (Figure 7.10) For Query1M, once again, we opt for requirement CC1, with the aim of cost containment whenever possible. Consequently, according to the consumers *low latency* requirement, we propagate *normal* to all nodes.

Regarding the update frequency, *MERGE* node N7 needs to combine three streams, two of which are *continuous*, while one is *periodic*. As a result, the merge node is divided into two parts: one for combining the *continuous* flow before performing partial computations, and the other for aggregating the results with those from the periodic part. It's worth noting that the final merge operation, highlighted in yellow in the diagram, will also involve a sorting step while combining periodic and continuous data.

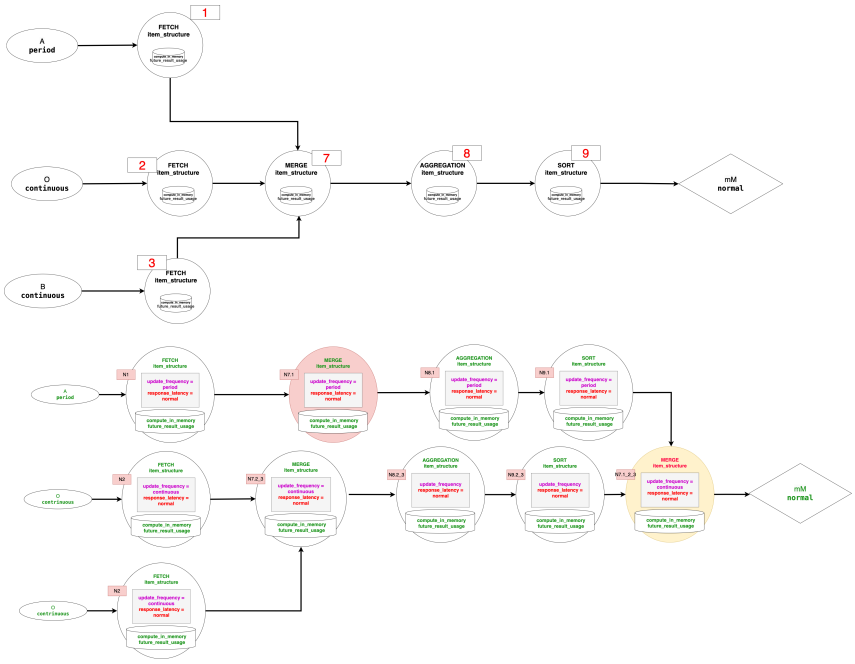


Figure 7.10: Query1M Parameters according to CC1

.

Query2M (Figure 7.11) Regardless of the chosen requirement, since the destination has a *low latency* requirement, as in the case of Query1B, we propagate this requirement to all nodes. Regarding the update frequency, as all sources are *continuous*, we propagate *continuous*.

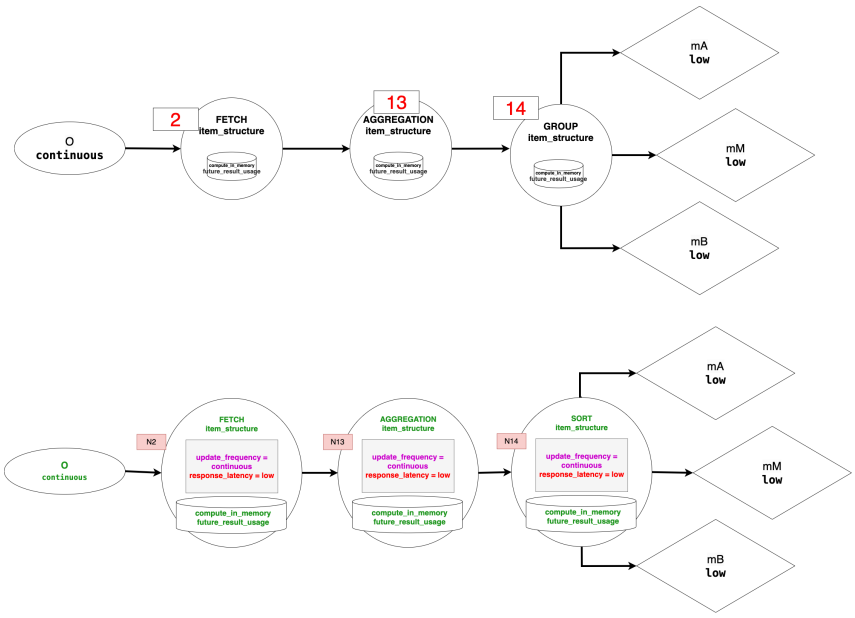


Figure 7.11: Query2M Parametrization.

.

8 | Methodology: Algorithm

This chapter describes in details the methodology's algorithm, which has been introduced in Chapter 6. Its input is a `dataflows`, whose elements are the `dataflows`, each of which has the structure of the input described in Chapter 7. For each `dataflow`, the methodology guides the architect to instantiate a `model_instance`, with a structure described in Section 8.1 describes the structure of the logical model, whereas in Section 8.2 a detailed description of the methodology is provided.

8.1. Model Instance

This section shows the instance of the model for the methodology application, which we define as `model_instance`. Its structure is shown in Figure 8.1.

```
model_instance {
    ingestion_component = no            // || [ || ingestion_component ]
    merge = no                          // || [ || merge ]
    processing_component = no           // || [ || processing_component ]
}
```

Figure 8.1: Instance of the Methodology.

It is composed of an attribute for each component that composes the logical model. At the beginning, all attributes are set to `no`, to indicate that the initial instance is empty. Then, during the implementation of the methodology, if a logical component must be activated, the corresponding attribute is assigned to the correspondent list. According to Chapter 6, each instance of the model have a list of *Ingestion* components, one of *Merge* components, and one of *Processing* components. These list can either be empty or contain one or more elements.

This is translated in the instance structure by considering a list of instances for the `ingestion_component`, for the `merge_component` and for the `processing_component`, which can either be empty or contain one or more instances.

Figure 8.2 shows the structure of the instance for each component. Each component has its own features, represented in the structure as attributes initially set to `no` and modified during the methodology implementation.

```
ingestion_component {
    node_id
    producer_id

    data_acquisition = no               // || data_acquisition
    processing_request = no             // || processing_request
    processing_response = no            // || processing_resonse

    unaltered_storage = no              // || storage || periodic
    process_in_memory = no              // || yes
}

merge_component {
    data_acquisition = no               // || data_acquisition
    processing_request = no             // || processing_request
    processing_response = no            // || processing_resonse

    parallelism = no                    // || parallelism
    extensions = no                     // || extensions

    transformed_storage = no            // || storage || periodic
    process_in_memory = no              // || yes

}

processing_component {
    data_acquisition = no               // || data_acquisition
    processing_request = no             // || processing_request
    processing_response = no            // || processing_resonse

    processing_actions = no             // || processing_actions

    query_knowledge = no                // || query_knowledge
    language = no                       // || language
    parallelism = no                    // || parallelism
    extensions = no                     // || extensions

    derived_storage = no                // || storage
}
```

Figure 8.2: Components' Instances.

Each features has its own instance (Figure 8.3), which is instantiated during the implementation. Also in this case, the initial instance has all attributes set to `no`. These are turned to `yes` if required during the methodology.

```
// istanze delle components features

data_acquisition {
    single-item = no        // || yes
    multi-item = no         // || yes
}

processing_request {
    static = no             // || yes
    dynamic = no            // || yes
}

processing_response {
    on-demand = no          // || yes
    continous = no          // || yes
}

query_knowledge {
    known = no              // || yes
    unknown = no            // || yes
}

processing_actions {
    filtering = no          // || yes
    aggregating = no        // || yes
    grouping = no           // || yes
    joining = no            // || yes
    sorting = no            // || yes
    pattern_matching = no   // || yes
}

language {
    declarative = no        // || yes
    imperative = no         // || yes
    oop = no                // || yes
    functional = no         // || yes
}

extensions {
    temporal = no           // || yes
    geographical = no       // || yes
}

parallelism{
    data = no               // || yes
    task = no               // || yes
}
```

Figure 8.3: Computation Features Instances.

Each component has its storage too. For each component, this is initially set to `no`, and optionally the methodology can instantiate a `storage` instance. The instance of the storage is shown in Figure 8.4, and it includes the storage's features initially set to `no` and replaced when needed by the methodology with their related instance.

```
storage {
    persisted_data_type         // unstructured || semistructured || drived
    atomicity = no              // || atomicity
    isolation = no              // || isolation
    availability = no           // || availability
    scalability = no            // || scalability
    durability = yes
}
```

Figure 8.4: Storage's Instance.

Notice that `durability` is initially set to `yes`. This is because the durability capability is needed in any storage. Figure 8.5 shows instances for features related to the storage part, whose attributes, initially set to `no` could be turned `yes` during the application of the methodology.

```
perisited_data_type {
    structured = no             // || yes
    unstructured = no           // || yes
    semi-structured = no        // || yes
}
atomicity{
    global-transaction = no     // || yes
    local-transaction = no      // || yes
    eventual-transaction= no    // || yes
}

isolation {
    strong = no                 // || yes
    weak = no                   // || yes
}

availability {
        highly = no             // || yes
        available = no          // || yes
}

scalability {
    vertical = no           // || yes
    horizontal = no         // || yes
}
```

Figure 8.5: Storage Features Instances.

8.2. Detailed Pseudo-code of the Methodology

This section outlines the two main steps of the methodology. It begins with defining the architecture using a Directed Acyclic Graph (DAG), following the completion of the preliminary steps described in Chapter 7. At this point, we have the different data flows with the necessary parameters to initiate the actual methodology.

The methodology is divided into three main parts: in the first part, as described in Section 8.2.1, we map the data flows to the model by instantiating a model instance for each data flow. Subsequently, for each data flow, we identify the types of systems with which they can be implemented (Section 8.2.2), and finally, we combine all the systems to obtain the set of systems to implement the architecture and create the infrastructure (Section 8.2.3).

8.2.1. STEP 1: From Data-Flows to Model Mapping Intances

The first crucial step in our methodology involves instantiating a model for each data flow. Each node within the data flow corresponds to an instance of a model component. "Fetch" actions correspond to instances of ingestion components, "merge" actions to instances of merge components, while other actions fall under data processing actions.

We begin with the *FETCH* nodes, which must determine their `data_acquisition` method: if the data producer has a continuous update frequency, they acquire data with a single-item capability; if the associated producer has a periodic update frequency, they employ the data acquisition multi-item capability. The same principle applies to *MERGE* nodes. In accordance with the query flow definition, even if there are multiple input nodes, all will share the same update frequency, which is used to identify the data acquisition mode of the merge component.

Subsequently, following the criteria illustrated in Figure 8.6, the methodology assigns the component the abilities related to `data_acquisition` and `processing_response` functionalities. Furthermore, optional features are activated. The `processing request` is part of the optional features; however, it is automatically activated as static, as the data-flow we are modeling is a static query, meaning that it known at implementation time. If the architect wishes to include dynamic queries on the data produced by a node, they must specify it among the optional features.

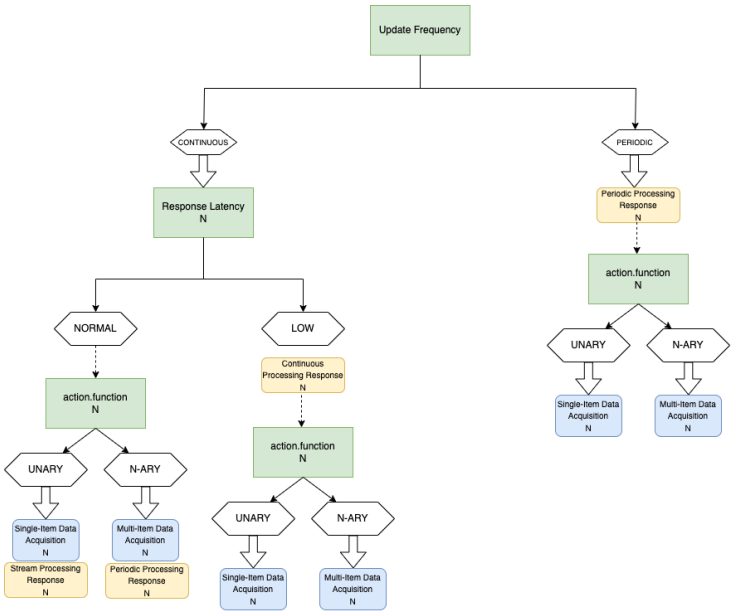


Figure 8.6: Computation Features Mapping.

Finally, the methodology checks if that node requires any type of storage and assigns it, if necessary, based on the criteria outlined in the Figure 8.7 schema.

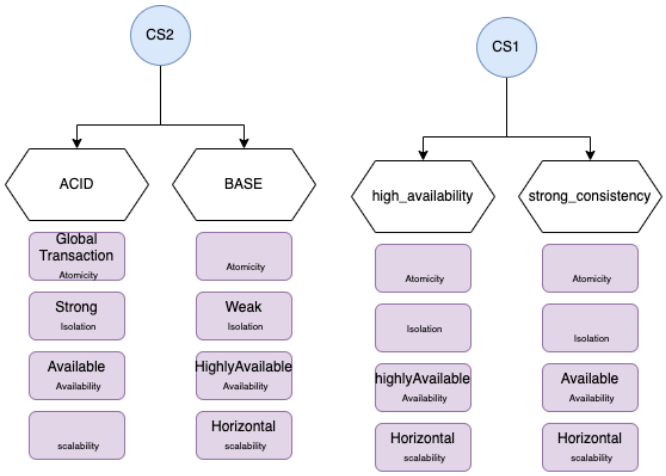


Figure 8.7: Storage Features Mapping.

The code displayed in Figure 8.8 represents the implementation of the first part of the methodology. The functions `activateStorage` and `activateOptional` are not included here, as they essentially consist of a series of "if" conditions, similar to those used in computational components.

```
// dataflow is a list of nodes
methodology1(dataflows) {
    models = [];
    for each dataflow in dataflows:
        flow-instances = [];
        for node in dataflows
            flow-instances.add(from-node-to-instance(node));
        models.add(flow-instances)
    return flow-instances;
}


// TO DO FOR EACH NODE OF EACH DATA FLOW.
from-node-to-instance (node) {
            //check type
    if ( node.action == fetch)
        instanceM = new ingestion_component
        assignDataAcquisiton(instanceM);

    else if (node.action == merge)
        instanceM = new merge_component;
        assignDataAcquisiton(instanceM);

    else
        instanceM.action = new processing_component

    // PERIODIC UPDATE FREQUENCT
    if (node.update_frequency == periodic) {
            instanceM.processing_response.periodic = yes
            instanceM.data_acquisition = multi-item
    }
    else {     // CONTINUOUS UPDATE FREQUENCY
        if (node.response_latency == low) {
            instanceM.response_processing.continuous = yes

            if (node.action.function == unary )
            instanceM.data_acquisition = single-item
            else
            instanceM.data_acquisition = multi-item
        }
        else if (instanceM.response_latency == normal) {
            if (node.action.function == unary) {
                instanceM.data_acquisition = single-item
                instanceM.processing_response = continuous
            }
            else if (node.response_latency == low) {
                instanceM.data_acquisition =multi-item
                instanceM.processing_response = periodic
            }
        }
    }

        // facultative features to activate. //
    activateOptional(instanceM, node.optional-features);

        // storage & in-memory //
    if (node.compute_in_memory == yes)
        instanceM.process-in-memory = yes;

    if (future_result_usage = in_range)
        instanceM.storage.persist_in_memory = yes;

    if (future_result_usage == storage)
        activate_storage(instanceM, storage.requirement)

        return InstanceM
}
```

Figure 8.8: Code related to the first step of the methodology.

Continuation of the Example: Mapping the Data Flow within the Model

In the following, we only consider Query1A and Query1B. We assume that Query1B has a storage requirement on node N12 with BASE critera. Figure 8.9 shows the mappings of components for Query1A, while Figure 8.10 shows that for Query1B.

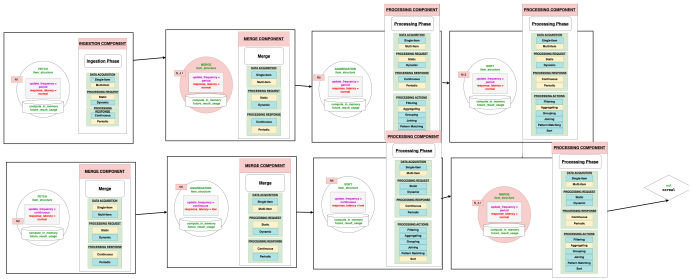


Figure 8.9: Mapping of Query1A within the model.

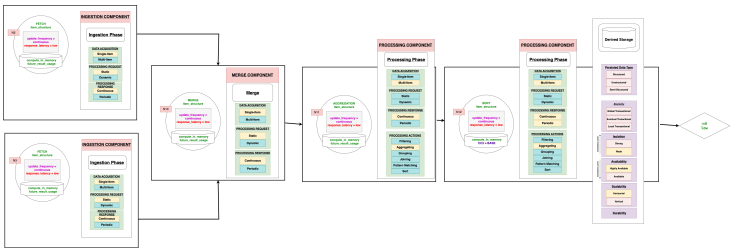


Figure 8.10: Mapping of Query1A within the model.

8.2.2. STEP 2: From Model Mapping Instances to Architecture Mapping

After the completion of the first step, we have a series of data-flows where the nodes are mapped to the various components of our model. The next step in the methodology involves merging the data flows to map the entire architecture with the model. In doing so, some components, specifically those present in multiple data flows, may be mapped to more than one instance of a model component.

Based on the characteristics of the different instances of a single architecture component, the next step will determine whether to split it into two systems or, if feasible, select a system that covers the features of both components.

Continuation of the Example: Merging the Data Flows to obtain the Model

Figure 8.12 illustrates how the components related to the Query1A and Query1B flows are integrated.

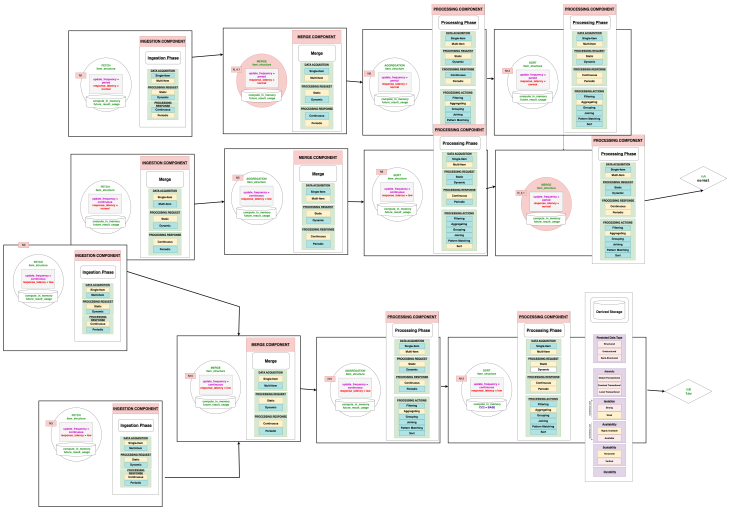


Figure 8.11: Mapping of Query1A and Query1B within the model.

Node two is present in both flows, and consequently, it has two different instances of the ingestion component, each activating different abilities. In the next step, we will decide how to manage them.

8.2.3. STEP 3: From Systems Data-Flows to Architecture Systems

Now we have a complete view of the infrastructure mapped onto our model, and we can proceed to identify the categories of systems for implementing the architecture.

The first thing to do is to distinguish between continuous queries and on-demand queries. In the graph we've modeled, these queries differ in their final node: continuous queries end with a consumer, while on-demand queries end with an internal node (processing or fetch). Consequently, in the latter case, we need to consider a state-centric component even if it's not explicitly requested by the architect. Furthermore, in this case, it's possible to implement the entire component with just one state-centric component.

In the second case, when queries end with external consumers and are thus continuous, the presence of a database is because the architect has explicitly requested it. However,

all components must also be associated with a data-centric component to facilitate the data flow of the query.

The first step is to group components with similar requirements because they represent actions that can be performed by the same system. For example, in Figure X, nodes N1, N_4.1, N5, and N12 represent actions that can be executed by the same technology since they have the same required abilities in terms of data acquisition and processing response. The same applies to N5 and N6 and N10 and N11.

Node N2 requires special attention as it is shared by two flows. However, it behaves the same way in both flows and can therefore be combined.

After consolidating the infrastructure, systems are selected. To select those related to data-centric, we refer to the criteria shown in Figure 8.12. We consider the Processing Response Feature, which is the capability to respond to a request from another component. If the methodology has periodic activation, it means the required processing response is periodic, and thus a batch technology is chosen. If it's continuous, a streaming technology is needed, and in the case of multi-item data acquisition, a stateful operator is also required to handle multi-item data.

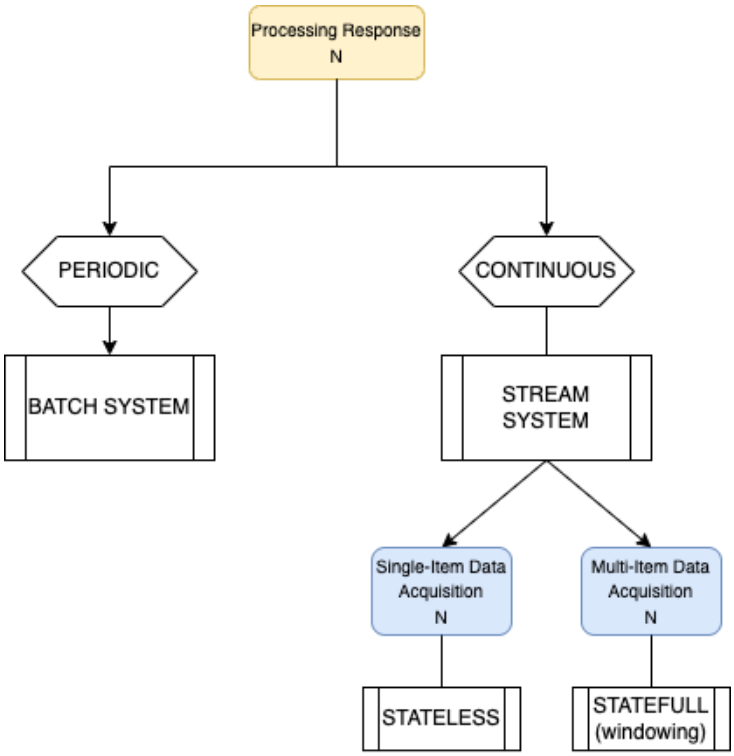


Figure 8.12: Mapping for State-Centric systems.

Regarding storage, the methodology guides the selection between a SQL database, a

NoSQL database, or a Distributed File System (DFS) based on the storage features. If no storage features are active, then it recommends a DFS. However, if the data is unstructured or semi-structured, a NoSQL database is suggested. For structured data, the methodology analyzes the mapped abilities. If highly availability and/or horizontal scalability are activated, it implies that these requirements are explicitly needed, making SQL technologies less suitable. If these requirements are not activated, the methodology suggests using a SQL database. At this point, when identifying a NoSQL database is necessary, it examines the corresponding node action. If the node is an "INGEST" node, a key-value database is sufficient since the data being saved is not processed in any way. If the data in a node corresponding to any action type lacks structure, then a key-value database might be the right solution. A "pattern matching" node indicates relationships within graphs, suggesting the use of a graph database. If the "extensions on temporal data" feature is enabled, a columnar database is recommended. If, after all these checks (implemented with conditionals), the methodology doesn't lead to a clear NoSQL solution, it provides the architect with a set of options for that component. In any case, once the methodology has offered plausible types of processing and storage systems, the architect should further investigate which specific systems of the identified type cover the secondary features. In other scenarios, the methodology can present various categories of options to the architect.

9 | Related Work

The main goal of this study is to try to find ways automate the decision-making process in the design of architectures for distributed systems, in response to specific requirements that may vary depending on use cases. This automation could bring significant benefits in the context of Infrastructure as Code (IaC). Infrastructure as Code (IaC) is the practice of automatically configuring system dependencies and provisioning both local and remote instances. In this context, to automate the provisioning of infrastructure, developers adhere to a specific syntax and write configurations in a manner similar to that of software source code [48].IaC and its effective orchestration demand models and descriptions capable of encompassing abstract, structural, and behavioral aspects. These models should be generic enough to cover not only how the infrastructure should behave but also service property specifications, enabling the orchestrator to replace services while preserving service properties ([54]).

Given its definition, it fits well within a context of an automated methodology that, starting from requirements, identifies a potential solution for designing an architecture. The result of this technology could serve as input for an IaC tool that automatically constructs the infrastructure based on the methodology's output. In this case, the designer would only need to specify the requirements. This is emerging in more recent literature: in ([54]), they assert that the idea of finding a paradigm for requirements engineering, focusing on logically-related abstractions that gradually reduce the abstraction level among software requirements of any kind and with respect to any constraint, is gaining momentum. However, due to the complexity of the systems to be modeled, we are still far from finding a complete solution.

For example, TOSCA could be a language that would benefit from such a methodology since it allows the design and programming of infrastructure designs using a standard notation and language. TOSCA provides reusable nodes and edges to specify infrastructure as a graph or topology ([21]).

Furthermore, as also addressed in the introduction, research has long investigated distributed database architectures, providing architectural patterns to follow when specific

requirements arise. Additionally, architectures of large distributed systems have been studied with the goal of highlighting reusable architectural patterns. However, modern literature lacks both a comprehensive overview that attempts to encompass all aspects of various data-intensive systems and a systematic methodology that could be beneficial, especially when combined with the world of IaC and frameworks like TOSCA.

10 | Conclusions and future developments

In summary, this research has dedicated its first part to the analysis of the theoretical aspects of big data system architectures, with the aim of creating a generalizable model capable of adapting to a wide range of real-world scenarios. Subsequently, we examined how these theoretical concepts find practical application in various systems, attempting to categorize them based on their abilities in both data processing and storage. The second part of this work focused on implementing these concepts to develop a decision-making algorithm for technology selection based on specific requirements. The methodology allows for a certain flexibility in the selection of decision criteria, also opening up the possibility of customized solutions by the architect with minor modifications. Furthermore, it adapts to the architect's needs based on the technologies to be used, suggesting categories and allowing them to choose based on their own personal knowledge. Given its systematic nature, one could contemplate implementing the procedure and offering a framework where the architect inputs the requirements and receives suggestions. Additionally, there is potential for a more in-depth exploration of feature selection to further enhance the coverage of various data-intensive architectures.

Bibliography

[1] Cassandra documentation. URL https://cassandra.apache.org/_/index.html.

[2] Couchdb documentation.

[3] Mongosb documentation.

[4] Redis documentation.

[5] Apache flink documentation.

[6] Geomesa documentation. URL https://www.geomesa.org/.

[7] Apache storm documentation.

[8] Hibernate documentation. URL https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html.

[9] Apache kafka stream documentation.

[10] URL https://docs.oracle.com/en-us/iaas/mysql-database/doc/getting-started.html.

[11] mysqldump documentation. URL https://dev.mysql.com/doc/refman/8.0/en/mysqldump.html.

[12] Oracle documentation.

[13] Postgis documentation, . URL https://postgis.net/docs/manual-3.4/.

[14] Postgresql documentation, .

[15] Apache spark documentation. URL https://spark.apache.org/docs/latest/index.html.

[16] Apache storm documentation.

[17] Timescale documentation. URL https://docs.timescale.com.

[18] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*, pages 67–78, 2000. doi: 10.1109/ICDE.2000.839388.

[19] T. Akidau, P. Barbier, I. Cseri, F. Hueske, T. Jones, S. Lionheart, D. Mills, D. Pauliukevich, L. Probst, N. Semmler, D. Sotolongo, and B. Zhang. What's the difference? incremental processing with change queries in snowflake. *Proc. ACM Manag. Data*, 1(2), jun 2023. doi: 10.1145/3589776. URL `https://doi.org/10.1145/3589776`.

[20] A. Ali and M. Abdullah. A survey on vertical and horizontal scaling platforms for big data analytics. *International Journal of Integrated Engineering*, 11, 09 2019. doi: 10.30880/ijie.2019.11.06.015.

[21] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, and D. A. Tamburri. Devops: Introducing infrastructure-as-code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 497–498, 2017. doi: 10.1109/ICSE-C.2017.162.

[22] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow.*, 7(3):181–192, nov 2013. ISSN 2150-8097. doi: 10.14778/2732232.2732237. URL `https://doi.org/10.14778/2732232.2732237`.

[23] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. HAT, not CAP: Towards highly available transactions. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*, Santa Ana Pueblo, NM, May 2013. USENIX Association. URL `https://www.usenix.org/conference/hotos13/session/bailis`.

[24] L. Belcastro, F. Marozzo, and D. Talia. Programming models and systems for big data analysis. *International Journal of Parallel, Emergent and Distributed Systems*, 34(6):632–652, 2019.

[25] E. Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2): 23–29, 2012. doi: 10.1109/MC.2012.37.

[26] R. Casado and M. Younas. Emerging trends and technologies in big data processing. *Concurrency and Computation: Practice and Experience*, 27(8):2078–2091, 2015.

[27] A. Davoudian and M. Liu. Big data systems: A software engineering perspective. *ACM Computing Surveys (CSUR)*, 53(5):1–39, 2020.

[28] P. M. Dhulavvagol, V. H. Bhajantri, and S. Totad. Performance analysis of dis-

tributed processing system using shard selection techniques on elasticsearch. *Procedia Computer Science*, 167:1626–1635, 2020.

[29] M. Diogo, B. Cabral, and J. Bernardino. Consistency models of nosql databases. *Future Internet*, 11(2):43, 2019.

[30] H. Garcia-Molina and K. Salem. Sagas. *ACM Sigmod Record*, 16(3):249–259, 1987.

[31] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba. Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In *2019 IEEE international conference on software maintenance and evolution (ICSME)*, pages 580–589. IEEE, 2019.

[32] J. Han, H. E, G. Le, and J. Du. Survey on nosql database. In *2011 6th International Conference on Pervasive Computing and Applications*, pages 363–366, 2011. doi: 10.1109/ICPCA.2011.6106531.

[33] S. Hanov, A. Roth, and H. Smit. Refactoring mysql to support materialized views. *University of Waterloo Waterloo, Ontario www. cs. uwaterloo. ca/ hsmit/reports/cs746—report. pdf*, pages 1–11.

[34] W. K. Hauger and M. S. Olivier. The role of triggers in database forensics. In *2014 Information Security for South Africa*, pages 1–7, 2014. doi: 10.1109/ISSA.2014.6950506.

[35] R. Hecht and S. Jablonski. Nosql evaluation: A use case oriented survey. In *2011 International Conference on Cloud and Service Computing*, pages 336–341, 2011. doi: 10.1109/CSC.2011.6138544.

[36] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan. A survey of distributed data stream processing frameworks. *IEEE Access*, 7:154300–154316, 2019.

[37] M. Kleppmann. *Designing Data-Intensive Applications*. O'Reilly, Beijing, 2017. ISBN 978-1-4493-7332-0. URL `https://www.safaribooksonline.com/library/view/designing-data-intensive-applications/9781491903063/`.

[38] R. Kumar Kaliyar. Graph databases: A survey. In *International Conference on Computing, Communication & Automation*, pages 785–790. IEEE, 2015.

[39] E. Levy and A. Silberschatz. Distributed file systems: Concepts and examples. *ACM Comput. Surv.*, 22(4):321–374, dec 1990. ISSN 0360-0300. doi: 10.1145/98163.98169. URL `https://doi.org/10.1145/98163.98169`.

[40] J. R. Lourenço, V. Abramova, M. Vieira, B. Cabral, and J. Bernardino. Nosql databases: A software engineering perspective. In *New Contributions in Information Systems and Technologies: Volume 1*, pages 741–750. Springer, 2015.

[41] M. Maier, A. Serebrenik, and I. Vanderfeesten. Towards a big data reference architecture. *University of Eindhoven*, pages 1–144, 2013.

[42] A. Margara, G. Cugola, N. Felicioni, and S. Cilloni. A model and survey of distributed data-intensive systems. *ACM Computing Surveys*, 56(1):1–69, 2023.

[43] A. B. M. Moniruzzaman and S. A. Hossain. Nosql database: New era of databases for big data analytics - classification, characteristics and comparison, 2013.

[44] R. Myung, H. Yu, and D. Lee. Optimizing parallelism of big data analytics at distributed computing system. *International Journal on Advanced Science, Engineering and Information Technology*, 7(5):1716–1721, 2017.

[45] A. Nayak, A. Poriya, and D. Poojary. Type of nosql databases and its comparison with relational databases. *International Journal of Applied Information Systems*, 5 (4):16–19, 2013.

[46] P. Pääkkönen and D. Pakkala. Reference architecture and classification of technologies, products and services for big data systems. *Big data research*, 2(4):166–186, 2015.

[47] M. M. Patil, A. Hanni, C. H. Tejeshwar, and P. Patil. A qualitative analysis of the performance of mongodb vs mysql database based on insertion and retriewal operations using a web/android application to explore load balancing — sharding in mongodb and its advantages. pages 325–330, 2017. doi: 10.1109/I-SMAC.2017.8058365.

[48] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams. A systematic mapping study of infrastructure as code research. *Information and Software Technology*, 108:65–77, 2019.

[49] H. Röger and R. Mayer. A comprehensive survey on parallelization and elasticity in stream processing. 52(2), apr 2019. ISSN 0360-0300. doi: 10.1145/3303849. URL `https://doi.org/10.1145/3303849`.

[50] O. Rusu, I. Halcu, O. Grigoriu, G. Neculoiu, V. Sandulescu, M. Marinescu, and V. Marinescu. Converting unstructured and semi-structured data into knowledge. In *2013 11th RoEduNet International Conference*, pages 1–4, 2013. doi: 10.1109/RoEduNet.2013.6511736.

[51] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010. doi: 10.1109/MSST.2010.5496972.

[52] M. Strohbach, J. Daubert, H. Ravkin, and M. Lischka. Big data storage. *New Horizons for a Data-Driven Economy: A Roadmap for Usage and Exploitation of Big Data in Europe*, pages 119–141, 2016.

[53] A. Struckov, S. Yufa, A. A. Visheratin, and D. Nasonov. Evaluation of modern tools and techniques for storing time-series data. *Procedia Computer Science*, 156:19–28, 2019.

[54] D. A. Tamburri, W.-J. Van den Heuvel, C. Lauwers, P. Lipton, D. Palma, and M. Rutkowski. Tosca-based intent modelling: goal-modelling for infrastructure-as-code. *Sics software-Intensive cyber-Physical systems*, 34:163–172, 2019.

[55] T. D. Thanh, S. Mohan, E. Choi, S. Kim, and P. Kim. A taxonomy and survey on distributed file systems. In *2008 Fourth international conference on networked computing and advanced information management*, volume 1, pages 144–149. IEEE, 2008.

[56] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, pages 1–12, 2012.

[57] Q.-C. To, J. Soto, and V. Markl. A survey of state management in big data processing systems, 2018.

[58] P. Viotti and M. Vukolić. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys (CSUR)*, 49(1):1–34, 2016.

[59] S. Wöllauer, D. Zeuss, P. Magdon, and T. Nauss. Rsdb: an easy to deploy open-source web platform for remote sensing raster and point cloud data management, exploration and processing. *Ecography*, 44(3):414–426, 2021.