



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

A Model and a Methodology for Data-Intensive Architectures

TESI DI LAUREA MAGISTRALE IN
ENGINEERING OF COMPUTING SYSTEMS- INGEGNERIA INFOR-
MATICA

Author: **Arianna Dragoni**

Student ID: 976071

Advisor: Prof. Alessandro Margara

Co-advisors:

Academic Year: 2022-23

Abstract

The thesis aims to offer data-intensive architects a systematic methodology for selecting a suitable set of technologies to implement a data-intensive architecture. Data-intensive systems architectures are software architectures to store and/or process large-scale data, according to their functional and non-functional requirements. As a result, the specific architecture design can vary, focusing on the key features it must have for effective data management and processing. This leads to the need to highlight all possible characteristics that data-intensive architectures can exhibit and to connect them to the requirements that an architect can gather in the initial design phase. In the first part of the thesis, a model encompassing all possible characteristics is provided. The model represents the data life cycle inside the architecture, which is divided in phases. Then for each phase the possible features that could be required are listed (both for storing and processing purposes). The methodology first uses the provided model to select necessary features considering the requirements, and then selects the set of technologies that allow to cover all the features and build up the architecture.

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

Abstract in lingua italiana

La tesi si propone di offrire agli architetti di sistemi data-intensive una metodologia sistematica per la selezione di un insieme adatto di tecnologie per implementare un'architettura data-intensive. Le architetture dei sistemi data-intensive sono architetture software progettate per memorizzare e/o elaborare dati su larga scala, in base ai loro requisiti funzionali e non funzionali. Di conseguenza, il design specifico dell'architettura può variare, concentrandosi sulle principali caratteristiche che deve avere per una gestione ed elaborazione efficace dei dati. Ciò porta alla necessità di evidenziare tutte le possibili caratteristiche che le architetture data-intensive possono manifestare e di collegarle ai requisiti che un architetto può raccogliere nella fase iniziale di progettazione. Nella prima parte della tesi viene fornito un modello che comprende tutte le possibili caratteristiche. Il modello rappresenta il ciclo di vita dei dati all'interno dell'architettura, che è diviso in fasi. Quindi, per ciascuna fase, vengono elencate le possibili caratteristiche che potrebbero essere richieste (sia per scopi di archiviazione che di elaborazione). La metodologia utilizza prima il modello fornito per selezionare le caratteristiche necessarie in considerazione dei requisiti, e quindi seleziona l'insieme di tecnologie che consentono di coprire tutte le caratteristiche e costruire l'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	3
2 A Model for Data-Intensive Architectures	5
2.1 Components	5
2.1.1 Overview	5
2.1.2 Interactions	7
2.2 High-Level Model Description	9
2.2.1 Data-Centric vs State-Centric Components	11
2.2.2 High-Level Model's Mappings	12
2.2.3 Multiple Data-Flows	15
2.3 Features	17
3 Computation Features	19
3.1 Phase's Computation Features	20
3.1.1 Language	21
3.1.2 Query Behavior	22
3.1.3 Query Knowledge	22
3.1.4 Elaboration Type	23
3.1.5 Extension	24
3.1.6 Parallelism	25
3.1.7 Transforming Actions	26
3.1.8 Processing Actions	27

4	Storage Features	29
4.1	Persisted Data Type Feature	30
4.2	Guarantees Features	31
4.2.1	Availability	31
4.2.2	Atomicity	34
4.2.3	Isolation	37
4.2.4	Durability	39
4.2.5	Scalability	40
5	Physical Components	43
5.1	State-Centric Systems	43
5.1.1	Database Systems	43
5.1.2	Others State-Centric Systems	44
5.1.3	Components Mappings	45
5.2	Data-Centric Systems	55
5.2.1	Processing Systems	55
5.2.2	Components Mappings	56
6	Methodology Overview	61
7	Methodology: Input Requirements	67
8	Methodology: Algorithm	71
8.1	Logical Model Instance	71
8.2	Detailed Pseudo-Code of the methodology	74
	Bibliography	79
	List of Figures	81

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. They refer to software infrastructures that comprise systems with distinct primary functions, including databases, queuing systems, storage systems, and analytics systems, among others, which collaborate to optimize data processing and management. These architectures can be regarded as an ensemble of components meticulously designed in accordance with the specific requirements of the company. Their purpose is to delineate the data life-cycle and generate essential information to meet the company's needs. According to the specific requirements, different components can be selected: both the type of data collected and the way in which it has to be managed and/or analyzed to extract useful information can vary a lot according to the needs of the company.

We consider a wide spectrum of requirements, encompassing both non-functional and functional aspects. Functional requirements are intricately linked to the nature of queries originating from external users, whereas non-functional requirements aims in safeguarding the integrity, consistency, scalability, and latency performance of our distributed environments. Choosing distributed environments instead of centralized ones introduces additional complexities in meeting these requirements, which translate into new requirements. 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 storage, and those that combine these functions. When software architects apply the methodology to a specific architecture, they are initially guided to choose from the model the characteristics relevant to that specific architecture, based on the list of 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 requirements 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. A set of requirements lead to the selection of a set of model's characteristics;
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 the model with its components and relative features. 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 the literature there are either lots of descriptions of single systems that try to cover as many requirements as possible, or specific architectures sometimes designed around a specific physical component, without exhaustively specifying the requirements they cover. However, as it will be detailed in Chapter ??, there is no complete model that can be adapted to every architecture based on the specific requirements it has to cover.

As a first main contribution of this thesis, we have created a model that tries to group all the characteristics that a *data-intensive processing and management architecture* can have. In fact, we will show that our model can be adapted both to architectures focused on data storage (i.e., databases), and to architectures focused on data processing. By selecting the *logical components*, we will show that the model can be adapted both to various architectures known in the literature, such as Lambda, Kappa, Liquid,.. and to systems such as OLAP, OLTP, Data Warehouse, Data Lake, ... In addition, the model can also be adapted to application-specific scenarios, which present more stringent requirements in terms of how the data is to be used and persisted.

Moreover, in the literature there is no systematic methodology from the point of view of software engineering: given a specific application scenario, there is no formal process to identify a suitable architecture and a suitable set of technologies to implement it. With this work, as a second main contribution, we also try to find a methodology to create an application-specific *data-intensive architecture*, starting from the requirements previously identified by the architect and building the physical infrastructure. Our methodology can be seen as an algorithm that the architect can follow to implement his application-specific architecture, giving it the requirements as input and obtaining as output a set of technologies to implement it.

Moreover, the methodology could be useful for orchestrators languages such as toasca.

2 | A Model for Data-Intensive Architectures

This chapter describes our model of a data-intensive architecture in terms of the logical components it implements. The model describes the data life-cycle within an architecture and highlights all the potential capabilities that may be provided. We start by describing the single components and their interactions in section 2.1. Then, a high-level model description is provided in Section 2.2, followed by an overview on how capabilities of each components are represented within the model in Section 2.3.

2.1. Components

We define a *component* as a piece of the architecture with a precise role in the data management and/or processing. The architecture is constructed from a collection of these components, which all together define the data life-cycle within the architecture. The model comprises all the possible components that can be implemented within the architecture. Then, the specific architecture will be composed of some of them according to its requirements.

2.1.1. Overview

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 submit queries within the architecture and receive responses. We define query any request received within the architecture regarding the data that the architecture receives from producers and manages in order to generate a response. The data that passes through the model and is optionally persisted can be *raw*, *raw-formatted* or *derived*.

We refer to the data received from sources as *raw*, indicating that it has not undergone any form of modification. The raw data can be subjected to various operations, transforming it into a *raw-formatted* data. Both raw and raw-formatted 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 raw-formatted data can undergo processing 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 producers. Derived information, in this case, would be the computed average of these measurements by the infrastructure.

Each component within the model comprises a phase and an optional storage. A *phase* correspond to a step in the big data life-cycle, and the *storage* represent the ability of the component to persist data at that specific phase. A component is said to be *state-centric* when the *storage* is activated, *data-centric* otherwise: *state-centric* components are databases, mostly dedicated to persist information; *data-centric* components are processing components used to process the data without having to persist it. When the storage is present, it has its *Data Structure*, which can be *Transactional*, *Documental*, *Key-Value*, *Graph* or *Columnar*, whose differences will be detailed in Chapter 4.

As it will be clarified in the following chapters, the methodology guides the architect to decide first to activate some of the components based on the requirements he/she has collected, and secondly to decide whether to activate or not the storage. The activation of a component represents the presence of the corresponding phase in the data life-cycle within the architecture. The activation of the storage allow to maintain data at that phase. When the storage is activated, the methodology requires to select the Storage Data Structure too.

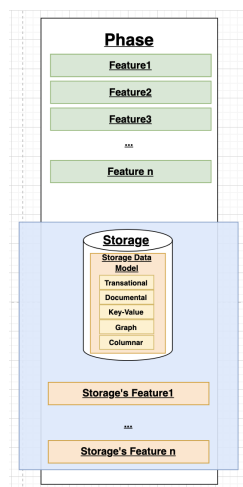


Figure 2.1: Logical Component Overview.

Both phase and storage within each component can have one or more features. We define

a *feature* as a capability that the architecture may or may not offer to cover a particular requirement. Features related to a phase are defined as *Computation Features*, whereas those related to the storage are defined as *Storage Features*. To explain how they differ, consider the component as an object in the object-oriented paradigm, which can have some methods (actions that can be performed over the data) and a state: a phase consists of a computation part, and a storage part. Methods are Computation Features, and the state is represented by Storage Features.

So, after activating the component (and optionally the storage), the methodology guides the architect to activate the features he/she needs according to the collected requirements.

As it will be more evident in the following chapters, the model presents parallels between storage features and computation features: there will be requirements that can be covered by both types of features. This is why the methodology requires to decide the type of logical component a-priori based on the requirements: State-Centric components require to activate the storage and to mostly select storage features, whereas Data-Centric components mostly require computation's features.

In the following, the *Storage Data Structure* is omitted from the figures for compactness. It will then be present again in the complete model shown in Chapter 6.

2.1.2. Interactions

The components within the model communicate with each other through the exchange of data and queries. 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 *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 *Query-Flow* as the process of transferring a query and its associated responses from one component to another.

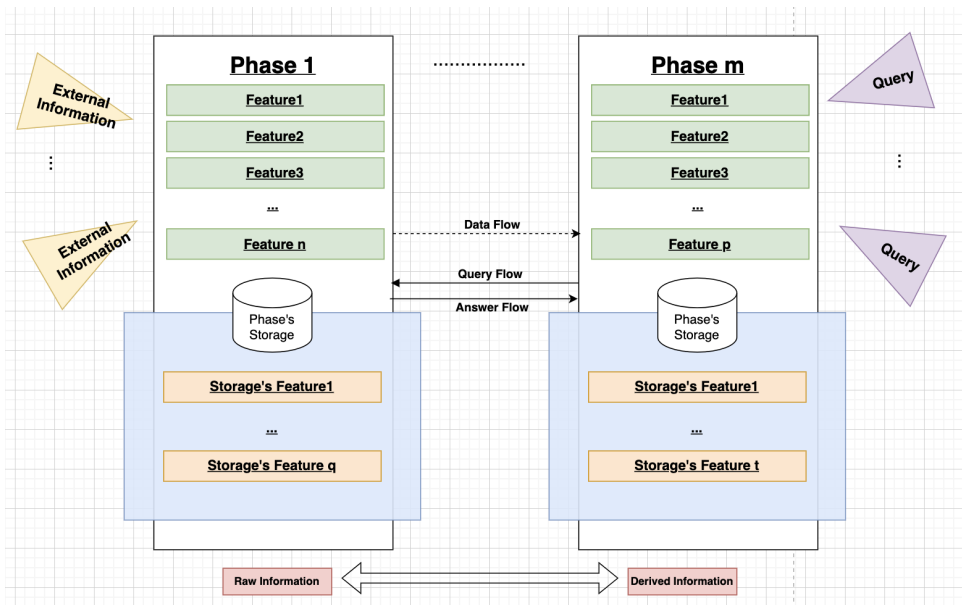


Figure 2.2: Phases' Interaction

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 cleaning operations (taking the form of *raw-formatted*), and then other information can be extracted starting from it by doing some processing: *derived* information can be produced (and optionally persisted). As a consequence, the model manages more raw information moving on the left components, and more elaborate information on the right components: *raw(-formatted)* information will be in the leftmost components, and *derived* one, when produced, in the rightmost components.

Data may flow from one phase to the other either in *push* mode or in *pull* mode: with *push* mode, the data is sent to the next phase with a time that can be configured by the designer; with *pull* mode, the receiver phase 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.

Query-Flow

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 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. The way in which information has to be analyzed and/or managed depends on the type of queries done over the infrastructure. Formally, queries can be *On-Demand* or *Continuous*: *On Demand* queries are activated when necessary, and as a consequence manage data in *pull mode*, whereas *Continuous Queries* are always active and process data when it is available, i.e., in *push mode*. In some cases, queries are not defined a-priori and require the entire processing at query time. In other cases queries are known upfront, and this allows to pre-compute answers to respond with low latency. Notice that the model here presents a gray area: a state-centric component with continuous queries known in advance may perform computations as soon as data arrives to pre-compute the answer. In this scenario, what we effectively obtain is a behavior similar to that of a data-centric component, as it processes the stream of entries into the database to pre-compute the changes.

Summing up, the model combines components to establish the data life cycle, with each component comprising a phase and optionally a storage. Data enters in the first component of the model from producers (external sources). Consumers send queries to the last component of the model. Components in between are used to save and process the information based on the incoming queries.

2.2. High-Level Model Description

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.

To describe the model, it is necessary to distinguish between *non-complex queries* and *complex queries*. To define them, we use the definitions of raw, raw-formatted, and derived data provided above. *Non-complex* queries requires wither raw or raw-formatted data, while complex queries require derived data. Summing up, *non-complex* queries ask for information that comes from producers, either as they ingest it (raw data) or in a different representation (raw-formatted). *Complex queries* ask for information that doesn't directly

originate from producers but necessitates the infrastructure to generate it through various processing actions.

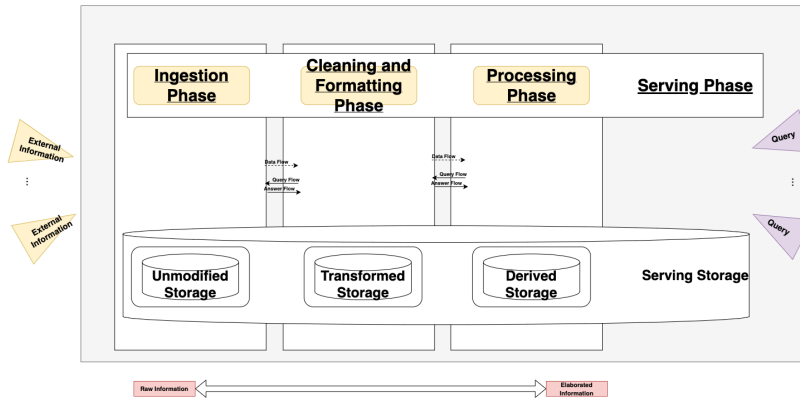


Figure 2.3: Model's Logical Components

Figure 2.3 represents the model, which is composed of four *phases* (i.e., *logical components*) with their correspondent storages.

- The *Ingestion Phase* 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 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 *Cleaning and Formatting Phase* acquires data from the previous phase. It is present if and only if some transformation over the raw data is necessary to facilitate the work that will be done by the following phases. The *Transformed Storage* can persist raw-formatted data produced in this phase.
- The *Processing Phase* is responsible for doing processing actions over the raw/raw-formatted information 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 Formatted storage is computed in this phase, and the average is derived data. The *Derived Storage* stores the derived information that is produced in this phase. It is used for in-memory purposes.
- The *Serving Phase* communicates with the consumers. It receives queries from the outside and sends the results back to the outside as soon as they are ready. It is used

for in-memory purposes. The *Serving Storage* can persist the results in-memory of the query.

Summing up, raw data can be stored in the *Unaltered Storage*. When data needs to be transformed before storage, it becomes raw-formatted and can be stored in the *Formatted Storage*. The *Transformed Storage* saves derived data.

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

2.2.1. Data-Centric vs State-Centric Components

The presence of a storage in a component indicates that (part of) the data resulting from that component is persisted in a database (i.e., the phase is a state-centric component). In the case of data-centric component, the storage is not used, and data flows to the next phase without being ‘remembered’. Consequently, it will no longer be available in the future. For example, if the processing phase is a data-centric component, the result of the query is directly sent to the serving 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 phase is a state-centric component, the result of the query is persisted in the transformed storage before being delivered to the serving phase.

The Serving Storage needs a further clarification. If the serving phase is a state-centric component, the serving storage persist data the previous phase send to it. This result may or may not exist in some other storage. If it does, it is additionally persisted in the serving storage; if not, it resides solely in the serving storage. The data persisted in the serving storage and also persisted in another storage consists only of the data necessary to respond to the query and may not encompass all the data present in the other storage.

Using the thermal measurement example, the system may compute the average over the thermal measurements, persisting some intermediate computations (e.g., the sum of the measurements and their count, that could be useful in future executions) if necessary. Assuming that data is stored as it arrives, as shown in Figure 2.4a, thermal measurements are stored in the unaltered storage, intermediate computations are in the transformed storage, and the average is both in the transformed Storage and in the serving Storage. If data needs some rearrangement, it passes through the cleaning and formatting phase

and is stored in the formatting storage. Notice that in this example the only possible query is the average. If other queries are possible, answers to those queries will be in the serving storage too: in the example, if another possible query requires to select a single measurement, as shown in Figure 2.4b, measurements will be in the serving storage.

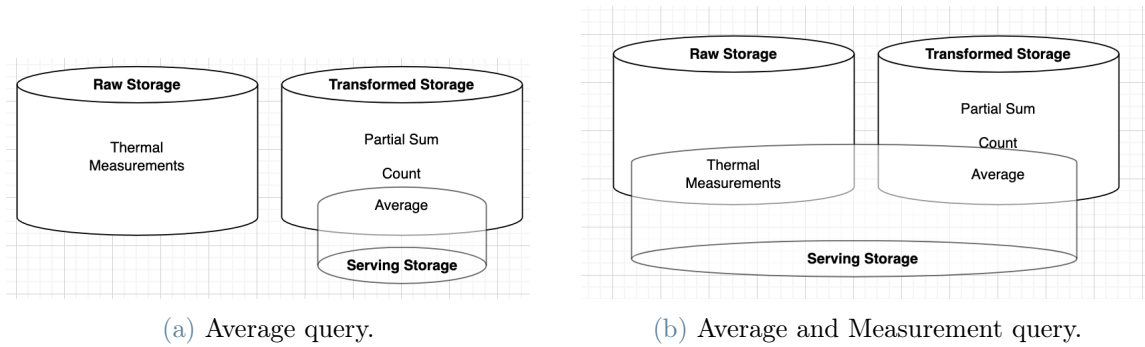


Figure 2.4: Example of usage of the storages.

2.2.2. 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.

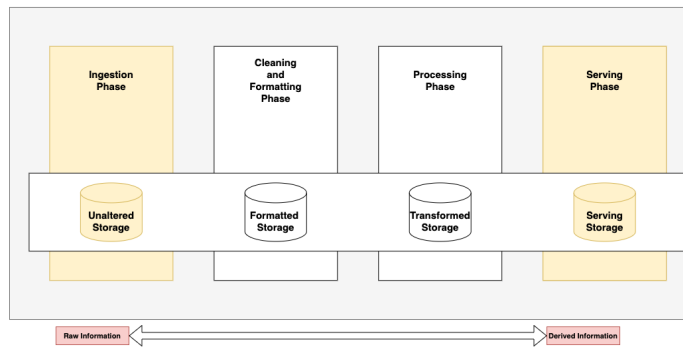


Figure 2.5: Example of Model's Mapping

An application of the model could be an infrastructure that gather raw data from producers, with receivers making requests for this raw data through non complex queries. The architecture of the infrastructure is mapped in the model by only activating the Ingestion Phase and the Serving Phase with their storages, as illustrated in Figure 2.5. The serving storage contains those raw data asked by the query. If the data arriving from producers still adhere to a specific schema or format, the Storage Data Model of the storages is selected accordingly.



Figure 2.6: Examples of Model's Mappings.

We assume the in-memory option both activated for storage and processing purposes. The cleaning and formatting phase is activated when raw data ingested by producers necessitate some form of transformation. In this scenario, the architect has the option to retain both the unaltered storage to store raw data and the formatted storage to store raw-formatted data or choose to keep only one of them based on the collected requirements. As an example, if queries only ask for raw-formatted data, the architect may decide to keep only the formatted storage (see Figure 2.6a). 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 formatted storage, it is possible to recompute the raw-formatted information starting from the raw one that is persisted in the unaltered storage. This last case is shown in Figure 2.6b. Furthermore, it might not be necessary to save the query result, which corresponds to the transformed data. In such cases, the architect can choose to persist only the raw data (Figure 2.6c) in the unaltered storage or to not persist anything at all (Figure 2.6c). Notice that in such scenarios the serving storage is not activated, as the result of the query is not persisted.

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



Figure 2.7: Examples of Model's Mappings.

Any time the query requires derived data that must be extracted through processing actions over the raw/raw-formatted data (e.g., aggregations to produce averages, sums, and more) the *Processing Phase* is activated. Whether to activate the transformed storage or not depends on the need to persist the derived information that has been obtained. This derived information can serve as either the answer to the query or as intermediate computations required to obtain the final result. Figure 2.6c illustrates a scenario in which the derived data, including both intermediate and final results, do not require persistence. As the final result is not persisted, the serving storage remains inactive. In the scenario depicted by Figure 2.6c, the final result does not need to be persisted, but intermediate computations do. Consequently, the transformed storage is activated to retain these intermediate computations, while the serving storage remains inactive, given that the final result is not persisted. Figure 2.6c describes the situation in which the final result needs to be persisted. In this case, both the transformed and the serving storage are activated. Whether they contain identical data or if the transformed storage retains additional data depends on whether the processing phase requires the persistence of intermediate results.

2.2.3. Multiple Data-Flows

Data-intensive architectures can represent infrastructures that need to handle different query-flows, which may require data to follow different data-flows based on the specific query requirements. As a result, the same architecture may need to simultaneously manage multiple data-flows, each corresponding to the different query-flow it handles. Given that our model describes the data-flow for a single query-flow, the overall architecture is depicted as multiple parallel instances of the model, each corresponding to a different data-flow/query-flow pair. 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 [9]:

- 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.

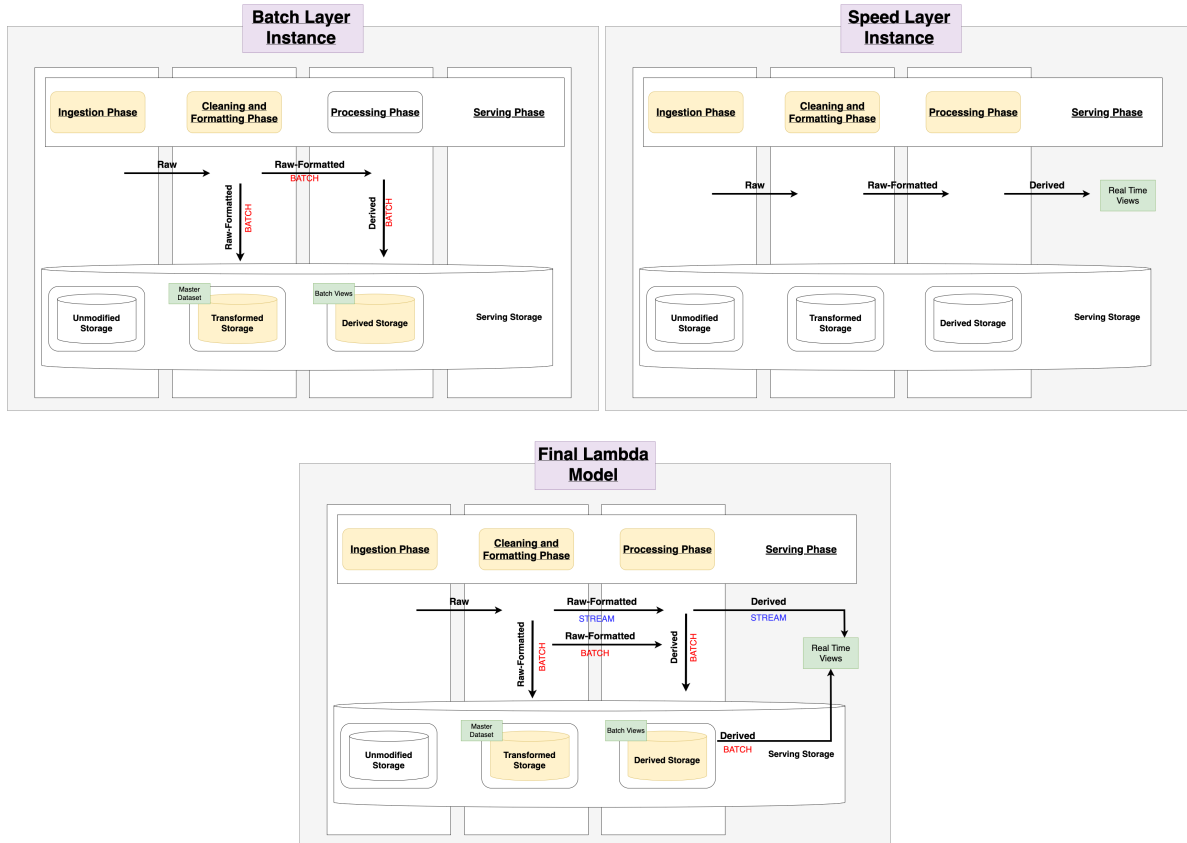


Figure 2.8: Lambda's Pipelines and Final Model

For this example, we consider that requirements are such that

- raw data needs to be transformed, and becomes raw-formatted data. If this is not the case, the *Cleaning and Formatting Phase* is omitted, and instead of the *transformed storage* we select the *unaltered storage*;
- only raw-formatted 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 2.8. 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 cleaning and formatting component. Here data is transformed in raw-formatted data and stored in the transformed storage, which corresponds to the master data-set. The processing phase periodically retrieve data from the transformed 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 cleaning and formatting component, which transforms it and directly produces real time views without persisting raw-formatted information. These two instances are merged to obtain the final model.

2.3. Features

Up until now, we've treated each component as a black box. Each component has a collection of *features* within, representing the potential capabilities it can have. The overall model, with its components containing the features, represents the complete set of potential capabilities that a data-intensive architecture can provide. We define *Computation* features as those that pertain to provide capabilities to process data within the architecture. While the majority of these features can be delivered by both data-centric and state-centric components, there are some that are exclusive to state-centric components. We define *Storage* features as those that refer to how data is stored and under which assumptions. These can be delivered only by state-centric components.

Components can have different abilities in providing each feature. In the context of computation features, these imply various actions that can be performed on the data. In the case of storage features, they relate to different characteristics that the storage can exhibit. A specific data-intensive architecture can be mapped to the model by selecting, for each data-flow, its specific components, and for each of these components select the features that are necessary to produce a concrete implementation.

As shown in Figure 2.9, features are represented as rectangles, and when the selection of the ability is required, the feature has rectangles to represent the different mechanisms inside the main one.

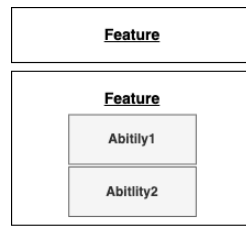


Figure 2.9: Feature Representations.

Chapter 3 and Chapter 4, provide a description of each feature; 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 computational aspect of model's components. Figure 3.1 provides an overview of the model, with a focus on Computation Features that encompass the architecture's capabilities for generating the data flow, while temporarily omitting storage features. Computation features are represented in green, while their respective abilities in blue. As it is illustrated, components may provide different capabilities based on their respective phases. Furthermore, the majority of these features are relevant to both state-centric and data-centric components, while some are exclusive to state-centric components' storage.

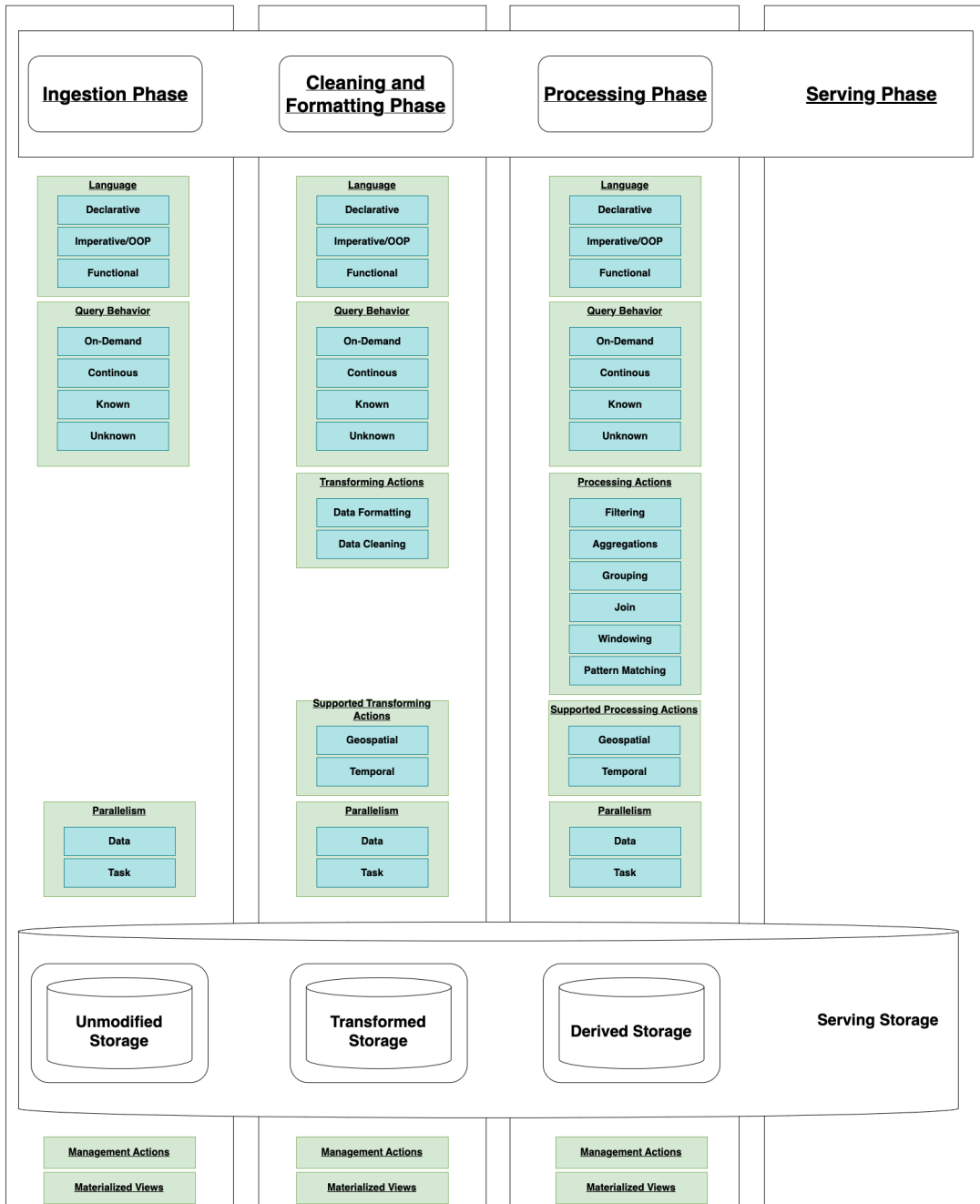


Figure 3.1: Computation Features Overview.

In the following sections, an explanation of each feature is provided, along with their respective abilities. Section 3.1 lists features of both state-centric and data-centric components, while Section ?? lists those exclusive of state-centric components.

3.1. Phase’s Computation Features

3.1.1. 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 contrast, in imperative and object-oriented languages (OOP), the emphasis is on the operations to be performed to achieve the desired result; users explicitly define a sequence of commands to be executed, and the system carries out these commands step by step. 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, which is present in each component, is depicted in Figure 3.2. Components can implement it using one or a combination of various programming languages. Within our model, the choice of language is guided by the selection of one or more of the following abilities: *declarative*, *imperative/OOP*, and *functional*.

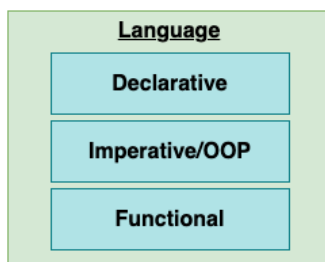


Figure 3.2: Language Feature.

In general, the architect who wants to make decisions based on the control of the execution flow chooses the imperative/OOP approach, while if he/she intends to specify the operations to be performed, he/she opts for a declarative approach. If, on the other hand, he/she wishes to define functions to enhance the language's expressiveness, he/she decides for a functional approach. Additionally, systems employing declarative solutions implement query optimization strategies that automate the selection of the most efficient execution for a given computation. This is possible because the system autonomously manages the execution process and can apply optimizations automatically. In contrast,

in alternative approaches, the system lacks the ability to predict the execution flow, and thus, optimization is the responsibility of the programmer. However, it should be noted that adopting declarative solutions may limit the language’s expressiveness to only what is provided by its syntax. Some systems mitigate this limitation by allowing developers to define custom functions (user-defined functions), thereby combining declarative and functional aspects to enhance the language’s flexibility. It is important to recognize that this strategy may reduce opportunities for automatic optimization since the system loses full control over execution.

3.1.2. Query Behavior

We define the *Query Behavior Feature* as the component’s capability to handle queries that may be triggered within the architecture at different points in time. We differentiate between two primary architectural abilities: responding to on-demand requests, triggered when needed, and addressing continuous requests, which activate as soon as new data enters the system. In our model, to emphasize the various abilities of architectures in responding to queries, we categorize query behaviors into *On-demand* and *Continuous*. Figure 3.3 illustrates the feature, which can be present in any component of the model.

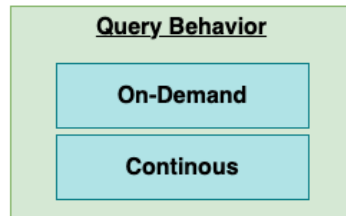


Figure 3.3: Query Behavior Feature.

We define *On-Demand Query Behaviour* the component’s ability to handle requests triggered when needed, and *Continuous Query Behaviour* the component’s ability to trigger queries as soon as data arrives.

3.1.3. 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 3.4 illustrates the feature, which can be present in any component of the model.

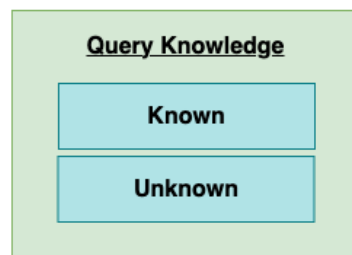


Figure 3.4: Query Behavior Feature.

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.

3.1.4. Elaboration Type

We define *Elaboration Type Feature* as the component's capability to elaborate data in various approaches. This feature can be found in components with any phase type, and the processing we refer to corresponds to that performed in the specific phase of the component. Viewing the data-set as a collection of items, one approach is to process one item at a time, while another approach involves gathering a group of items before starting the elaboration.

The feature's representation in our model is shown in Figure 3.5.

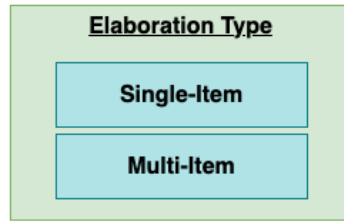


Figure 3.5: Elaboration Type Feature.

We distinguish between *Single-Item* and *Multi-Item* abilities in elaborating data. We define *Single-Item* as the ability to perform elaborations one item at a time, while *Multi-Item* as the ability to group items before elaboration. While the first approach does not require storage for processing, the second approach necessitates, at the very least, a means to temporarily store the items that need to be processed within the same group. This doesn't necessarily translate to the presence of storage in the model, but it could also manifest as less complex yet efficient temporary persistence techniques to provide this capability within the architecture. As a result, the architect, in order to enable multi-item elaboration, doesn't necessarily need to activate storage.

3.1.5. 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 seamlessly 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 3.6, where we differentiate between *Temporal* and *Geospatial* abilities for managing and/or processing data.

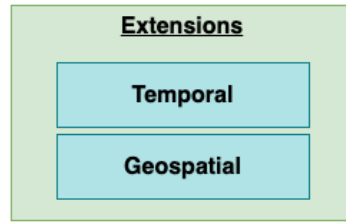


Figure 3.6: Extensions Feature.

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.

3.1.6. 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 3.7.

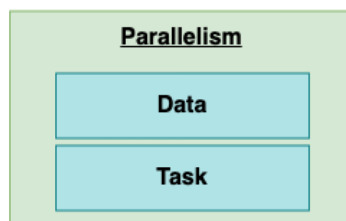


Figure 3.7: Parallelism Feature.

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.

3.1.7. Transforming Actions

We define *Transforming Actions Feature* as the capability of the component in handling actions for cleaning and formatting the data that comes from the outside. This feature is typical of the Cleaning and Formatting Phase, which takes data from the Ingestion Phase and prepares it for processing and/or storage. Transforming actions are those executed during the Transformation phase of the ETL (Extract, Transform, Load) process. The ETL process involves extracting data from various sources, processing it, and then storing it. Transformation includes actions like data cleaning, which aims to correct erroneous data and provide clean data by addressing issues such as missing data and rejecting invalid data, and Data conformance, which aims at ensuring data correctness and compatibility with other master data. The component with this capability should provide a set of data transformations or operators such as filtering, sorting, inner joins, and outer joins[12].

In our model, the feature in question is represented in Figure 3.8.

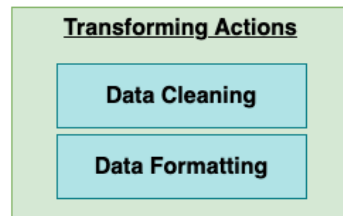


Figure 3.8: Transforming Actions Feature.

We distinguish abilities in providing operations to clean data (referred to as *Data Cleaning*) and to format data (referred to as *Data Formatting*). We define *Data Cleaning* as the component's ability to provide functions for cleaning data, such as removing duplicates, handling null values, and verifying integrity rules. We define as *Data Formatting* the component's ability to format data in a manner that aligns with the standards required by other components within the architecture, ensuring it can be processed and saved appropriately.

3.1.8. 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 3.9 depicts the feature's representation in our model, where we distinguish between *Filtering*, *Aggregation*, *Grouping*, *Joining*, *Pattern Matching* abilities.

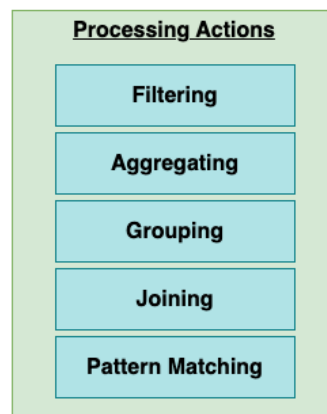


Figure 3.9: Processing Actions Feature.

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 *Pattern Matching* 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.

4 | Storage Features

This chapter describes the features each component can have when the storage is activated. We *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.

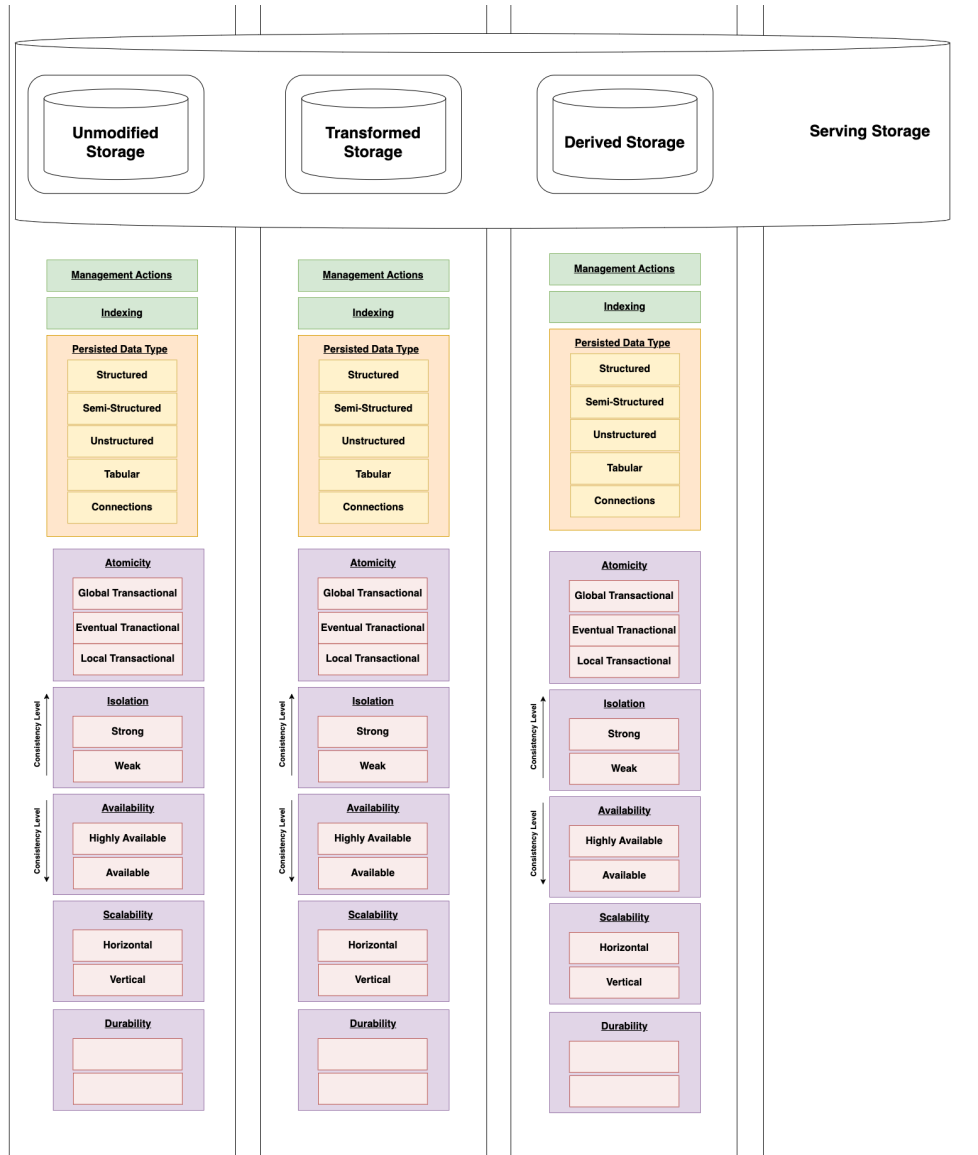


Figure 4.1: Storage Features Overview.

The following section outlines the various data types that can be stored within a storage system (Section 4.1), as well as the features associated with guarantees in terms of non-functional requirements (Section 4.2).

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, on the other hand, has a more flexible schema. 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.

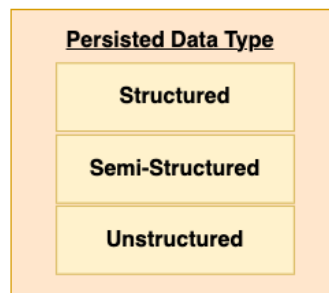


Figure 4.2: Persisted Data Type Feature.

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. Guarantees Features

4.2.1. 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[6]. 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 [13]. 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 [16]. 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 [13]. 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 [13]. 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*.

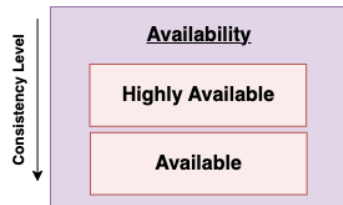


Figure 4.3: Availability Feature.

We define *Highly-Available* 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.

4.2.2. 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 [13].

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[13]. 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[13]; 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 [11]. 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.

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.

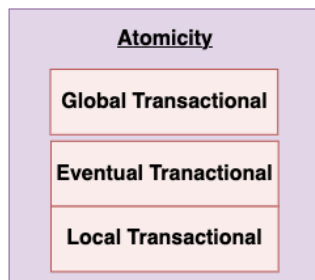


Figure 4.4: Atomicity Feature.

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.

4.2.3. 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[4]. 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 [13]. 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[4]. 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 [13]. To ensure serializability, a first requirement is a global transactional atomicity ability of the component (Subsection 4.2.2): 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[6]. Moreover, they can cause deadlocks that are usually solved automatically with some technique by the system[13]. 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[6].

Consequently, various weaker isolation levels have been introduced and categorized based on the race conditions they mitigate or avoid [4], and some of them can be implemented in highly-available scenarios[7].

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[13]. However, this approach necessitates the implementation of locking mechanisms to prevent dirty writes, making it unsuitable for high availability scenarios [6]. 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[13].

An example of isolation level that can be implemented with highly-availability is the Read Committed isolation level[7], 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[13].

Hence, it has been demonstrated that many systems can achieve highly-availability by relaxing serializability guarantees and allowing some race conditions to occur[7]. 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 isolation 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.

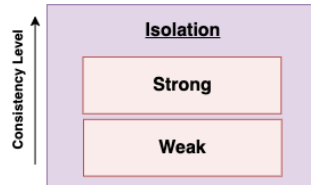


Figure 4.5: Isolation Feature.

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.

4.2.4. 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[13]. 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 implement it: Independent Checkpointing and Distributed Checkpointing. 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

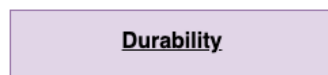


Figure 4.6: Durability Feature.

4.2.5. 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[13].

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[13].

Vertical scalability typically involves upgrading hardware components, such as CPU, RAM, or storage. The problem with a this approach is that the cost grows faster than linearly. Horizontal scalability, on the other hand involve split 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*

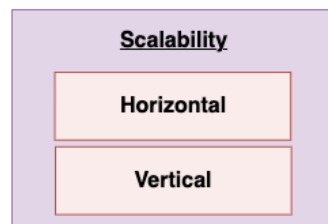


Figure 4.7: Scalability Feature.

5 | Physical Components

This chapter provides mapping between components and systems.

5.1. State-Centric Systems

5.1.1. Database Systems

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 on data consistency, providing the SQL language 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. SQL solutions are primarily vertically scalable, which means that to handling increased workloads, one must enhance the existing hardware (e.g., by adding a more powerful CPU), which can be economically costly. Additionally, they lack flexibility since the schema must be determined in advance and cannot be easily altered.

NoSQL databases 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: to increase the workload, it's sufficient to add additional machines and distribute tasks across multiple systems. Consequently, as data volume grows, the economic expense required is lower than with SQL databases.

Different NoSQL solutions are available, which differs in the way they persist data and in the query expressiveness they offer.

- *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.
- *Key-Value* NoSQL solutions associate to each item a key, and stores 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.

Based on this, various NoSQL database types are better suited for different query scenarios. For simple lookup queries, key-value solutions excel, while 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. Lastly, for queries primarily focused on analytics and aggregations, column-based databases emerge as the optimal selection.

Therefore, SQL solutions are chosen when there are stringent requirements in terms of data integrity, and there is the need to perform complex queries over this data. NoSQL solutions are better when the database must be easily scalable due to significant increases in the volume of data, and/or when it is not possible to define a schema on the data a priori.

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. We reviewed MySQL, Oracle, PostgreSQL for SQL databases, MongoDB, CouchDB, Cassandra, Redis, Neo4J, HBase for NoSQL.

5.1.2. Others State-Centric Systems

Distributed File Systems

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[14].

So, as databases are used for data management, allowing you to store, organize, and retrieve information, they are designed to ensure data reliability and durability through mechanisms like data replication or backup. 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.

However, they do not support ACID transactions and do not have consistency guarantees like databases. Additionally, they primarily handle unstructured or semi-structured data, such as files and documents, whose access is generally based on files and directories and does not support complex queries like databases.

In general, Distributed File Systems (DFS) are often an appropriate choice when you want to store data without the need for strict structuring or complex querying operations. DFS are particularly suitable for storing unstructured or semi-structured data, such as text files, images, videos, and documents, and are designed to provide a scalable and reliable storage mechanism for such data.

They are used with processing systems, and data is stored in a DFS for access by big data frameworks like Apache Hadoop or Apache Spark for processing and analysis.

5.1.3. Components Mappings

Computation Components Mappings

Language 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 [1], 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 al-

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.

Query Behavior 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. Therefore, if architects wish to support real-time continuous queries, it often requires integrating these databases with other technologies capable of handling this type of query behavior. As for key-value and graph databases, even though they do not directly support continuous queries, they can be part of a broader solution that includes streaming systems or real-time processing engines. In these complex architectures, the key-value or graph database is used to store data or intermediate results, while the streaming system takes care of the continuous analysis of real-time data. 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 or Triggers to define Stored Procedures, which can be seen as continuous queries. These Stored Procedures can be triggered in response to specific events and can analyze and modify data in real-time. 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.

Query Knowledge All databases are capable of responding to both known and unknown queries, but their ability to do so efficiently varies significantly depending on the type of database chosen. This variation arises from the fact that, for known queries, it is possible to optimize the architecture, indexes, and queries themselves to ensure high performance. However, for unknown queries, it is important to have a flexible and scalable database that can efficiently handle a wide range of requests without requiring a complete restructuring each time a new query is introduced. As a result, SQL databases are generally well-suited for situations where queries are anticipated and configured in advance, as they can be optimized to deliver high performance under these conditions. However, they tend to lose efficiency when confronted with unexpected queries, as they may require schema changes or significant adaptations to handle the new requests. On the other hand, NoSQL databases adapt well to both scenarios, as they are characterized by flexibility and scalability. They can be easily scaled to accommodate growing workloads and quickly

adapted to new requirements without the need for a complete database schema overhaul. This flexibility makes them suitable for both anticipated and unexpected queries, enabling greater agility in the evolution of the database system.

Elaboration Type All databases can perform both single-item and multi-item elaboration. In fact, they persist the data-set in their storage, and can retrieve items either one at a time or in group to perform elaborations. In the context of databases, elaborations may involve actions to be performed before storing the data through some insert mechanism and actions to be performed over data still stored in the database. In the first scenario, processing one item at a time entails either inserting a single item, while processing multiple items involves inserting multiple items at once. This insertion process may also involve data manipulation if it is handled internally by the database. In the second case, the first step in elaborating involves retrieving items that need to be processed using a query language from the database. Elaborating one item at a time in this case is performed using lookup queries, where a single item is retrieved by its primary key, while group processing is carried out through queries that retrieve more than one item through queries that filter elements within the data-set based on some attribute.

SQL databases use the `SELECT` or `UPDATE` clauses with a filter on the primary key to retrieve or manipulate a single item, and the same clauses with filters on other attributes to work with multiple items. Column-based databases can perform operations on individual data similarly to SQL, as they provide a similar language for data processing. However, they are primarily designed for complex processing on a group of data and are therefore preferable in multi-item scenarios. Key-value databases are primarily designed for single-data elaborations, using `get` and `set` commands for the primary key. Therefore, in multi-item scenarios, they may not be the optimal solution. Document-based databases can perform both single-item and multi-item operations. In fact, they provide methods for inserting, retrieving, updating, and deleting both individual items and groups of items. For example, MongoDB's query language provides methods such as `insertOne` or `updateOne` for single-item operations and methods like `insertMany` or `updateMany` for multi-item operations. Graph-based databases can perform both single-element and multi-element operations. For instance, the Cypher query language used in Neo4j allows you to run queries on both individual elements, whether nodes or relationships, and on graphs or subgraphs composed of multiple elements.

Extensions 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[3].

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[2].

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. 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, RSDB utilizes HBase as its underlying storage and is optimized for temporal data. Another example is ClickHouse, an open-source columnar database optimized for high-performance analysis, particularly well-suited for managing temporal data and time series. 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 is a spatial data extension that can be integrated with Cassandra or HBase.

Parallelism 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[15], 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 [10].

Transforming Actions In most SQL and NoSQL databases, it is common to find built-in support for data formatting actions, including conversion and string splitting, among other operations. These features allow you to handle data formatting directly within the database before saving it. For example, in MySQL, you can utilize clauses like `SELECT SUBSTRING_INDEX` to split a string, while MongoDB provides the ability to write scripts, such as those in JavaScript, to adapt data to the desired JSON format. The same principle applies to data cleaning actions. Databases often offer features for data cleaning and validation, including filtering, integrity constraint checks, duplicate removal, and other operations. However, it is important to note that despite these internal capabilities, the most common approach is to manage both data cleaning and data formatting operations outside the database. In this scenario, an external processing system prepares the data before saving it to the database. This approach provides greater flexibility in data processing and allows you to tailor formatting and cleaning operations to meet the specific project requirements.

Processing Actions As for transforming actions, 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. 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 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. 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 repeatable 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. For NoSQL databases, concurrency control is crucial. All databases of this type implement some mechanism to manage simultaneous access. Lock-based such as Two-Phase Locking are not commonly used in NoSQL databases. They are primarily employed in SQL databases to ensure ACID properties. However, some databases, such as MongoDB or Cassandra, may incorporate certain locking techniques. In general, document-based databases utilize a Timestamp-Based mechanism in which each document is associated with a timestamp relative to its last modification by a user (e.g., MongoDB). This is useful for resolving conflicts during write operations. Similarly, for Key-Value databases, the most common strategy is Timestamp-Based, where a timestamp is associated with each key-value pair. 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.

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 help in the implementation of the Sagas pattern to maintain data consistency in distributed transactions. For instance, 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 offer Replication, each with its own strategies. Looking at the documentations of various relational databases, it is possible to under-

stand 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 network. 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, 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. However, some allow configuration for synchronous or semi-synchronous propagation, such as MongoDB and Cassandra.

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. Most 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. Similar to SQL databases, the primary strategy for NoSQL databases involves implementing 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. The Document database MongoDB employs an append-only log for write operations. 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.

5.2. Data-Centric Systems

This section discusses systems suitable for data-centric components, which we refer to as *Data-Centric Systems*.

5.2.1. Processing Systems

We define *Processing Systems* those systems whose characteristics primarily focus on performing computations over data without persisting it over time. We categorize them into *Batch-Processing* systems and *Stream-Processing* Systems, with the former implementing batch processing and the latter executing stream processing. 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 [8]. 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 5.2.1 analyzes the two categories, highlighting differences and commonalities,

while Subsection 5.2.2 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. In our model, we distinguish between data coming from external producers or other data-centric components and data already present in storage. Batch technologies can handle data in both cases, as they can arrive either already divided into batches, for example, when a producer sends data in separate files, or they can arrive in a streaming fashion, as in the case of being transmitted by a stream-processing system. In the first case, computation is performed when the batch arrives, while in the second case, the batch-processing system implements some windowing to divide and process them. Stream-processing systems, on the other hand, require data to arrive in a streaming manner, so they need to be connected 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 [8].

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.

5.2.2. Components Mappings

Computation Components Mappings

Language Most processing systems 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, you 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.

Query Behavior In processing systems, queries are used to inquire about the processing performed by the system. In the context of batch-processing systems, queries are executed on data stored in separate batches. It is common for on-demand queries to be activated by users to obtain results derived from periodic processing. For example, if you wish to perform analyses on data collected within specific time windows, the batch process will be executed on each of these windows, and queries will be used to retrieve the desired results over time. In the case of stream-processing systems, processing occurs in real-time as data flows through the system. Operations on streaming data must be predefined in advance since the system processes data as it arrives, and therefore, traditional on-demand queries cannot be executed. Summing up, the usual approach is to use stream-processing systems to answer continuous queries and use batch-processing systems to handle on-demand queries.

Query Knowledge The design of data processing systems involves managing known queries, which are planned in advance to meet specific system requirements. However, flexibility is a key feature of many data processing systems, both batch and streaming, that enables them to successfully handle unknown queries as well. When a new unknown query or unexpected analysis requirement arises, it is possible to define a new data processing pipeline or a sequence of ad-hoc operations and integrate it into the system's code. This flexibility allows for extending the system to accommodate new analysis requests without the need for a complete overhaul of the existing infrastructure.

Elaboration Type In processing systems elaboration entails executing actions on items, either individually or in groups. In processing systems, it is common practice to elaborate items one at a time as they arrive in real-time within a stream-processing context. Conversely, in a batch-processing system, the elaboration of single items is atypical. Typically, data is elaborated in batches, where multiple elements are worked on simultaneously

within each batch. Elaborating single items in a batch context is uncommon and is regarded as inefficient in terms of resource utilization. Elaborating multiple items is a capability present in both types of processing systems. In batch-processing systems, this approach is the standard practice, as operations are carried out on entire batches that contain groups of items. In stream-processing systems, on the other hand, techniques such as windowing or data aggregation within specific intervals, which we define as stateful techniques, are employed to facilitate concurrent elaboration of multiple items.

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. 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 highlights 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[5], 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 simplifies 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.

Extensions 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 you 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 Data processing systems are primarily designed 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

In batch-processing systems, data parallelism involves the parallel processing of batches, while task parallelism pertains to the simultaneous execution of various independent operations on each batch. In stream-processing systems, data parallelism entails dividing the incoming stream and performing operations on each sub-stream in parallel before aggregating the results. Task parallelism, on the other hand, involves the concurrent execution of multiple independent operations or functions on each incoming data partition.

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. However, there are significant differences in how parallelism is managed between these two approaches. In the case of streaming, users often have greater control over defining parallelism. This is because streaming data is constantly evolving, and users can define data partitioning and how tasks are distributed to adapt to the specific needs of the application. In batch processing systems, automation of parallelism is more common, as batch processing involves the processing of data-sets in separate batches.

Transforming Actions Transforming actions, both data formatting and data cleaning, are typically managed through processing systems. These systems can encompass both

batch-processing systems like Apache Flink and stream-processing systems like Apache Kafka or Apache NiFi. The choice of a batch-processing system, which is advantageous in terms of computational resources, is usually made when dealing with large volumes of data that are periodically ingested into the system and require complex processing. Conversely, a stream-processing system is preferred when there is a need to perform computationally less complex transformations, such as simple data integrity checks, on continuously arriving streaming data.

Processing Actions Both batch-processing and stream-processing systems offers the capability to perform processing actions.

Batch-processing systems allow for the filtering of elements within a batch based on specific 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.

6 | Methodology Overview

This chapter describes the software methodology to create and implement a data-intensive architecture. The 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 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.

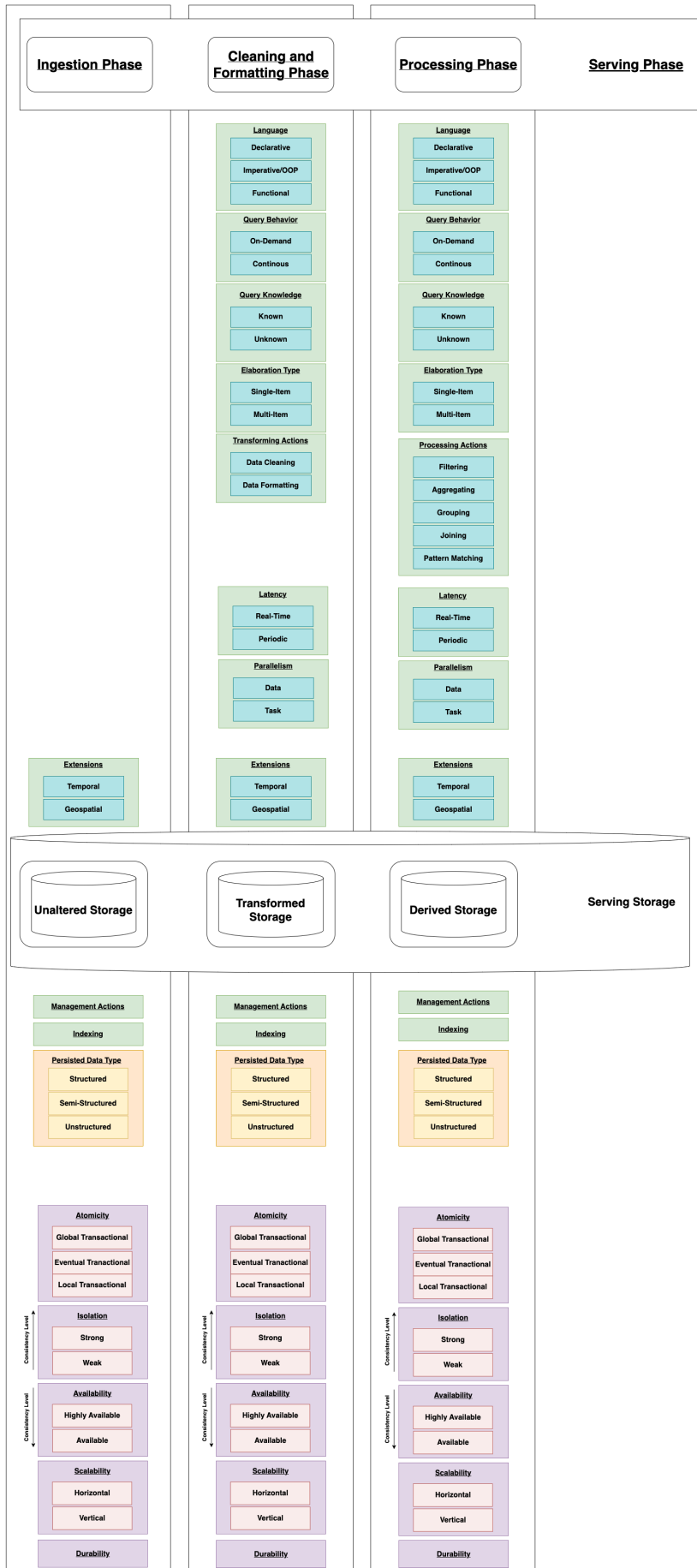


Figure 6.1: Complete model.

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 a list of data flows, whose size determines the number of instances of the model, as it is explained in Section 2.2.3. Data flows are 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.

activation comprende anche le abilities

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

```

methodology (methodology_input) {
    for dataflow in methodology_input:
        // Step0: Intantiate the model

        // Step1: Intantiate Ingestion Component
        // 1.1 : Intantiate Ingestion Phase
        // 1.2 : Optionally Activate In-Memory Option for the Phase
        // 1.3 : Optionally Intantiate Unaltered Storage
        // 1.4 : Optionally Activate In-Memory Option for the Storage

        // Step2: Optionally Intantiate Cleaning and Formatting Component
        // 2.1 : Intantiate Cleaning and Formatting Phase
        // 2.2 : Optionally Activate In-Memory Option for the Phase
        // 2.3 : Optionally Intantiate Transformed Storage
        // 2.4 : Optionally Activate In-Memory Option for the Storage

        // Step3: Analyse the Query Flow
        for subquery in query-flow:
            // 3.1 Intantiate one Processing Phase
            // 3.2 : Optionally Activate In-Memory Option for the Phase
            // 3.3 : Optionally Intantiate Derived Storage
            // 3.4 : Optionally Activate In-Memory Option for the Storage

        // Step4: select technologies for the dataflow

    // Step5: Merge technlogies to each flows and obtain the final
}

```

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 *Cleaning and Formatting* component is required when data has to be formatted 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 *Transformed* storage is required when when data needs some formatting 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. The *Ingestion* component is the first one to be instantiates. Moreover, if necessary, the storage is instantiated and in-memory processing and storage activated. The same happen for the *Cleaning and Formatting* component if necessary. 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. Given that sub-queries that make up the data flow's query have their own characteristics, they may require different computation and storage capabilities. As a consequence, an instance of the model can have more than a *Processing* phase, and each phase may or may not activate its storage to persist the result of that specific computation.

The next step is to select the appropriate set of systems to implement the architecture. Chapter 5 shows how to perform the *mapping*: technologies 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.

After carrying out all the steps for each data-flow, if there is more than one, the architect must merge the different instances, thus obtaining the final set of technologies, which is the output of the methodology.

7 | Methodology: Input Requirements

The input of the methodology are extracted from the information about producers, consumers, queries and requirements previously collected by the architect. Specifically, the architect has to know

- how and where is the data produced by producers
- the queries that are ingested by consumers.
- the requirements the architecture has to satisfy, both functional and non-functional.

Starting from this, the architect can produce the input of the methodology, which will be used to apply the algorithm described in Chapter 8.

As a first step, the architect must identify the data flows by categorizing the queries that the architecture should handle based on their individual requirements and data dependencies. This process is elaborated in Section 2.2.3.

The methodology relies on an input list of data flows tailored to the structure illustrated in Figure 7.1, which must be derived from the previously accumulated knowledge.

```

methodology_input // list of dataflow

dataflow {
  producers_list // list of producers
  consumers_list // list of consumers

  keep_unaltered = no // || storage_abilities
  keep_in_memory = no // || yes

  elaboration // no || elaboration
}
query_flow // no || query_flow

producer {
  id
  edge = no // || yes
  producer_data_type // [structured || unstructured || semistructured]
}

consumer {
  id

  elaboration {
    transformed_data_type // [ || structured || unstructured || semistructured ]
    transformed_actions // [ || data-cleaning || data-formatted ]
    computation_abilities // computation_abilities
    compute_in_memory // || yes
  }
  keep // no || storage_abilities
  keep_in_memory = no // || yes
}

query_flow {
  nodes_flow // list of processing_nodes
}

storage_abilities {
  atomicity // no || global || eventual || local
  isolation // no || strong || weak
  availability // no || highly_available || available
  scalability // [ || horizontal || vertical ]
}

computation_abilities {
  language // [ || declarative || imperative_oop || functional ]
  query_behavior // [ || on-demand || continuous ]
  query_knowledge // [ || known || unknown ]
  latency // [ real-time || periodic ]
  parallelism // [ || data || task ]
  extensions // [ || temporal || geospatial ]
}

node {
  processing_action // [ || filtering || aggregating || grouping || join || pattern_matching ]
  data_type // [ structured || unstructured || semi-structured ]
  compute_in_memory // no || yes
  computation_abilities // computation_abilities
  keep // no || storage_abilities
  in_memory_storage = no // || yes
}

```

Figure 7.1: Input Format.

Each `dataflow` includes a list of producers (`producers_list`) and a list of consumers (`consumers_list`), which identify the external actors of the data-flow. Additionally, it contains information regarding the management of incoming external data. The `keep_unaltered` attribute signifies the need to retain data in its original state, while the `keep_in_memory` attribute the necessity to store data in memory. The `elaborate` attribute indicates the need to perform some transformation on the incoming data. The `query-flow` represents the query that constitutes the data-flow.

Each of these attribute has its own instance, which the architect has to produce.

The list of producers is composed of one or more `producer` instances, whereas the list of consumers is composed of one or more `consumer` instances each of which has its own `id`. Additionally, each `producer` instance has also an attribute to identify the information they produce (`producer_data_type`), which can be `structured`, `unstructured`, `semistructured` or a combination of them, and an attribute to indicate whether they incorporate edge computing capabilities (`edge`).

The `keep_unaltered` instance is created when raw data needs to be persisted and corresponds to a `storage_abilities` instance where the where the storage capabilities are configured as `yes`.

The `storage_abilities` instance catalogs the non-functional capabilities of the storage. For each capability, the architect can select relevant abilities for the data-flow to indicate that specific requirements needs more attention. Regarding each attribute corresponding to a capability, in some cases the architect can choose zero, one, or a combination of abilities (`[|| ability || ability ...]`), in other cases it can choose either zero or one (`[ability || ability || ...]`).

The `elaboration` attribute is instantiated when incoming data to the system requires transformation. This attribute encompasses indications of the data type that should be produced (`transformed_data_type`), which can be `structured`, `unstructured`, `semistructured` or a combination of them, as well as the actions that need to be performed on the data (`transformed_actions`), which include `data-cleaning`, `data-formatted` or a combination of them. Similarly to `storage_abilities`, `computation_abilities` instance catalogs the capability that the component must expose for processing purposes with abilities which in some cases can be combined. Moreover, it includes information on whether to perform computations in memory (`compute_in_memory = yes`) or not (`compute_in_memory = no`). Also, an instance of `storage_abilities` could be instantiated for the `keep` attribute if the raw-formatted data needs to be persisted, and `keep_in_memory` attribute could be set to `yes` if this data has to be persisted in memory.

Finally, the `dataflow` exhibit the `query_flow`, which has a flow composed of a list of processing nodes. To produce this, the architect has to decompose the query in actions that needs to be performed and identify special requirements for each actions. Each action correspond to a processing node, which includes an attribute to identify the type of the action (`processing_action`) and one for the data type (`data_type`). Additionally, it includes an attribute to decide whether to compute in-memory (`compute_in_memory`) and to process in-memory (`in_memory_storage`). Finally, it includes one attribute to instantiate the `computation_abilities` and one to optionally instantiate the `storage_abilities` (`keep`.) Notice that the absence of the query flow indicates that only management actions (insert, updates, deletes) are allowed.

8 | Methodology: Algorithm

8.1. Logical 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
  cleaning_formatting_component = no // || cleaning_formatting_component
  processing_component = no         // || list of processing_component
  serving_component = no           // || serving_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 an instance, or a list of instances, of the phase. According to Chapter 6, each instance of the model have an *Ingestion* component, can have a *Cleaning and Formatting* component and a *Serving* component, and can have zero, one or more than a *Processing* component. This is translated in the instance structure by considering at most one instance for the `ingestion_component`, for the `cleaning_formatting_component` and for the `serving_component`, and a list of instances for the and `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 {
    extensions = no           // || extensions
    unaltered_storage = no   // || storage
}

cleaning_formatting_component {
    language = no           // || language
    query_behavior = no     // || query_behavior
    query_knowledge = no    // || query_knowledge
    elaboration_type = no   // || elaboration_type
    transforming_actions = no // || transforming_actions
    latency = no            // || latency
    parallelism = no        // || parallelism
    extensions = no         // || extensions

    transformed_storage = no // || storage
}

processing_component {
    language = no           // || language
    query_behavior = no     // || query_behavior
    query_knowledge = no    // || query_knowledge
    elaboration_type = no   // || elaboration_type
    elaborating_actions = no // || elaborating_actions
    latency = no            // || latency
    parallelism = no        // || parallelism
    extensions = no         // || extensions

    derived_storage = no    // || storage
}

serving_component {
    ingestion_phase = no    // || yes
    cleaning_formatting_phase = no // || yes
    processing_phase = no   // || yes
    unaltered_storage = no // || yes
    transformed_storage = no // || yes
    derived_storage = no    // yes
}

```

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.

```

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

language {
  declarative = no   // || yes
  imperative_oop = no // || yes
  functional = no    // || yes
}

query_behavior {
  on-demand = no    // || yes
  continuous = no   // || yes
}

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

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

transforming_actions{
  data-cleaning = no // || yes
  data-formatting = no // || yes
}

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

latency{
  real-time = no     // || yes
  periodic = 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 {
  management-actions = yes
  indexing = no           // || yes
  atomicity = no         // || atomicity
  isolation = no         // || isolation
  availability = no      // || availability
  scalability = no       // || scalability
  durability = yes
}

```

Figure 8.4: Storage's Instance.

Notice that `management-actions` and `durability` are initially set to `yes`. This is because the durability and management actions capabilities are needed in any storage. Figure 8.9 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.

```

atomicity{
|   global-transaction = no           // || yes
|   local-transaction = no          // || yes
|   eventual-transaction= no        // || yes
}

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

availability {
|   highly = 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 describes in details the methodology's algorithm, which has been introduced in Chapter 6. It takes as input a list of `dataflow` (whose structure has been described in Chapter 7), each corresponding to a different query. Then, for each data-flow, the methodology instantiate the model and activates the necessary components and features by instantiating the corresponding instances. The model instance will then be used to select the set of technologies necessary to implement the current query.

The following subsections describes the step to be performed for each data-flow.

```

methodology (methodology_input) {
  list df_systems = [];

  for dataflow in methodology_input:
    // Step0: Intantiate the model
    model_instance = new model_instance;

    // Step1: Intantiate Ingestion Component
    // 1.1 : Intantiate Ingestion Phase
    model_instance.ingestion_component = new ingestion_component;
    // 1.2 : Optionally Activate In-Memory Option for the Phase
    if (dataflow.keep_in_memory == yes)
      model_instance.serving_component.ingestion_phase = yes;
    // 1.3 : Optionally Intantiate Unaltered Storage
    if (dataflow.keep_unaltered != no)
      model_instance.ingestion_component.unaltered_storage = new storage;
      // activate storage abilities
      activateStorageAbilities(model_instance.ingestion_component.unaltered_storage,
                             dataflow.keep_unaltered);
    // 1.4 : Optionally Activate In-Memory Option for the Storage
    if (dataflow.keep_in_memory == yes)
      model_instance.serving_component.unaltered_storage = yes;

    // Step2: Optionally Intantiate Cleaning and Formatting Component
    if (dataflow.elaboration not empty)
      // 2.1 : Intantiate Cleaning and Formatting Phase
      model_instance.cleaning_formatting_component = new cleaning_formatting_component;
      activateLaborationAbilities (model_instance.cleaning_formatting_component, dataflow.elaboration);
      // 2.2 : Optionally Activate In-Memory Option for the Phase
      if (dataflow.elaboration.compute_in_memory == yes)
        model_instance.serving_component.cleaning_formatting_phase = yes;
      // 2.3 : Optionally Intantiate Transformed Storage
      if (dataflow.elaborating_actions.keep not Empty)
        model_instance.cleaning_formatting_component.transformed_storage = new storage;
        // activate storage abilities
        activateStorageAbilities(model_instance.cleaning_formatting_component.transformed_storage,
                                dataflow.elaborating_actions.keep);
      // 2.4 : Optionally Activate In-Memory Option for the Storage
      if (dataflow.elaborating_actions.keep_in_memory == yes)
        model_instance.serving_component.transformed_storage = yes;

    // Step3: Analyse the Query Flow
    for subquery in query-flow:
      // 3.1 Intantiate one Processing Phase
      model_instance.processing_component APPEND new processing_component@processing_component;
      // 3.2 : Optionally Activate In-Memory Option for the Phase
      if (subquery.compute_in_memory == yes)
        model_instance.serving_component.processing_component = yes;
      // 3.3 : Optionally Intantiate Derived Storage
      if (subquery.keep not Empty)
        activateStorageAbilities(pricessing_component.derived_storage,
                                subquery.keep);

      // 3.4 : Optionally Activate In-Memory Option for the Storage
      if (subquery.in_memory_storage == yes)
        model_instance.serving_component.derived_storage = yes;

    // Step4: select technologies for the dataflow
    df_systems APPEND elaborateTechnologySet(model_instance);

  // Step5: Merge technologies to each flows and obtain the final
  finalMerge(df_systems);
}

```

Figure 8.6: Main Methodology.

```

activateStorageAbilities(storage, storage_abilities) {
  // ATOMICITY
  if(storage_abilities.atomicity CONTAINS global)
    storage.atomicity.global-transaction = yes;
  if(storage_abilities.atomicity CONTAINS eventual)
    storage.atomicity.eventual-transaction = yes;
  if(storage_abilities.atomicity CONTAINS local)
    storage.atomicity.local-transaction = yes;

  // ISOLATION
  if(storage_abilities.isolation CONTAINS strong)
    storage.isolation.strong = yes;
  if(storage_abilities.isolation CONTAINS eventual)
    storage.isolation.weak = yes;

  // AVAILABILITY
  if(storage_abilities.availability CONTAINS hihgly_available)
    storage.availability.highly = yes;

  // SCALABILITY
  if(storage_abilities.scalability CONTAINS horizontal)
    storage.scalability.horizontal = yes;
  if(storage_abilities.scalability CONTAINS vertical)
    storage.scalability.vertical = yes;
}

```

Figure 8.7: Storage Abilities Activation.

```

activateElaborationAbilities (cleaning_formatting_phase, abilities) {
  // LANGUAGE
  if(abilities.language CONTAINS declarative)
    cleaning_formatting_phase.language.declarative = yes;
  if(abilities.language CONTAINS imperative_oop)
    cleaning_formatting_phase.language.imperative_oop = yes;
  if(abilities.language CONTAINS functional)
    cleaning_formatting_phase.language.functional = yes;

  // QUERY BEHAVIOR
  if (abilities.query_behavior CONTAINS on-demand)
    cleaning_formatting_phase.query_behavior.on_demand = yes;
  if (abilities.query_behavior CONTAINS continous)
    cleaning_formatting_phase.query_behavior.continuous = yes;

  // QUERY KNOWLEDGE
  if (abilities.query_knowledge CONTAINS known)
    cleaning_formatting_phase.query_knowledge.known = yes;
  if (abilities.query_knowledge CONTAINS unknown)
    cleaning_formatting_phase.query_knowledge.unknown = yes;

  // LATENCY
  if (abilities.latency CONTAINS real-time)
    cleaning_formatting_phase.latency.real-time = yes;
  if (abilities.latency CONTAINS periodic)
    cleaning_formatting_phase.latency.periodic = yes;

  // PARALLELISM
  if (abilities.parallelism CONTAINS data)
    cleaning_formatting.parallelism.data = yes;
  if (abilities.parallelism CONTAINS task)
    cleaning_formatting.parallelism.task = yes;

  // EXTENSIONS
  if(abilities.extensions CONTAINS temporal)
    cleaning_formatting.extensions.temporal = yes;
  if(abilities.extensions CONTAINS geographical)
    cleaning_formatting.extensions.geographical = yes;
}

```

Figure 8.8: Elaboration Abilities Activation.

```

activateProcessingAbilities (processing_phase, abilities) {
  // LANGUAGE
  if(abilities.language CONTAINS declarative)
    cleaning_formatting_phase.language.declarative = yes;
  if(abilities.language CONTAINS imperative_oop)
    cleaning_formatting_phase.language.imperative_oop = yes;
  if(abilities.language CONTAINS functional)
    cleaning_formatting_phase.language.functional = yes;

  // QUERY BEHAVIOR
  if (abilities.query_behavior CONTAINS on-demand)
    cleaning_formatting_phase.query_behavior.on_demand = yes;
  if (abilities.query_behavior CONTAINS continuous)
    cleaning_formatting_phase.query_behavior.continuous = yes;

  // QUERY KNOWLEDGE
  if (abilities.query_knowledge CONTAINS known)
    cleaning_formatting_phase.query_knowledge.known = yes;
  if (abilities.query_knowledge CONTAINS unknown)
    cleaning_formatting_phase.query_knowledge.unknown = yes;

  // LATENCY
  if (abilities.latency CONTAINS real-time)
    cleaning_formatting_phase.latency.real-time = yes;
  if (abilities.latency CONTAINS periodic)
    cleaning_formatting_phase.latency.periodic = yes;

  // PARALLELISM
  if (abilities.parallelism CONTAINS data)
    cleaning_formatting.parallelism.data = yes;
  if (abilities.parallelism CONTAINS task)
    cleaning_formatting.parallelism.task = yes;

  // EXTENSIONS
  if(abilities.extensions CONTAINS temporal)
    cleaning_formatting.extensions.temporal = yes;
  if(abilities.extensions CONTAINS geographical)
    cleaning_formatting.extensions.geographical = yes;
}

```

Figure 8.9: Processing Abilities Activation.

Bibliography

- [1] Hibernate documentation. URL https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html.
- [2] Postgis documentation. URL <https://postgis.net/docs/manual-3.4/>.
- [3] Timescale documentation. URL <https://docs.timescale.com>.
- [4] 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.
- [5] 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>.
- [6] 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>.
- [7] 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>.
- [8] R. Casado and M. Younas. Emerging trends and technologies in big data processing. *Concurrency and Computation: Practice and Experience*, 27(8):2078–2091, 2015.
- [9] A. Davoudian and M. Liu. Big data systems: A software engineering perspective. *ACM Computing Surveys (CSUR)*, 53(5):1–39, 2020.
- [10] 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.
- [11] H. Garcia-Molina and K. Salem. Sagas. *ACM Sigmod Record*, 16(3):249–259, 1987.
- [12] A. Kabiri and C. Dalila. Survey on etl processes. *Journal of Theoretical and Applied Information Technology*, Vol. 53, 07 2013.
- [13] 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/>.
- [14] 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>.
- [15] 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 retrieval 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.
- [16] P. Viotti and M. Vukolić. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys (CSUR)*, 49(1):1–34, 2016.

List of Figures

2.1	Logical Component Overview.	6
2.2	Phases' Interaction	8
2.3	Model's Logical Components	10
2.4	Shorter caption	12
2.5	Example of Model's Mapping	12
2.6	Shorter caption	13
2.7	Shorter caption	14
2.8	Lambda's Pipelines and Final Model	16
2.9	Feature Representations.	18
3.1	Computation Features Overview.	20
3.2	Language Feature.	21
3.3	Query Behavior Feature.	22
3.4	Query Behavior Feature.	23
3.5	Elaboration Type Feature.	24
3.6	Extensions Feature.	25
3.7	Parallelism Feature.	25
3.8	Transforming Actions Feature.	26
3.9	Processing Actions Feature.	27
4.1	Storage Features Overview.	30
4.2	Persisted Data Type Feature.	31
4.3	Availability Feature.	34
4.4	Atomicity Feature.	36
4.5	Isolation Feature.	39
4.6	Durability Feature.	40
4.7	Scalability Feature.	41
6.1	Complete model.	62
6.2	Methodology pseudo-code.	64

7.1	Input Format.	67
8.1	Instance of the Methodology.	71
8.2	Components' Instances.	72
8.3	Computation Features Instances.	73
8.4	Storage's Instance.	73
8.5	Storage Features Instances.	74
8.6	Main Methodology.	75
8.7	Storage Abilities Activation.	76
8.8	Elaboration Abilities Activation.	76
8.9	Pocessing Abilities Activation.	77