Phoenix

big/fast data processing pipeline

Whitepaper

Version 2.0, 25-MAR-2019
Carlos Godinho (cma.godinho@gmail.com)

1. Table of Contents

2. Introduction

Phoenix is an architecture concept introduced to establish new ideas for the creation of data pipelines applied to **Decision Support Systems** (DSS). It presents the challenges and possible solutions for the most common data patterns addressed in Telco business.

In Telco markets, DSS are subject to large data amounts, classified as **Big Data** solutions. As data volumes keep increasing, traditional techniques and patterns for data processing are no longer applicable.

Markets are also pushing non-functional requirements. Requesting systems with virtually zero latency. This type of solutions called **Fast Data**, provide real-time processing.

Phoenix architecture positions itself has a **future concept** to tackle Telco big and fast data challenges, being able to proper use system resources in acceptable timeframes and controlled costs. At the same time, new technologies and procedures are made available by industry. Understanding and designing over such technologies is the recommend away to redefine the acceptable procedures and push forward.

The next points start by presenting the challenges and common data patterns under discussion. They are followed by an architecture proposal and a detailed presentation of tools and techniques able to provide a practical solution.

3. Data Unit & Patterns

Telco DSS systems are feed with standardized data units and it is possible to identify common data processing patterns. Usually data received by DSS systems is already subject to transformations, presenting itself in an abstract layer above Telco protocols. This point describes the data structure and the applied processing patterns.

3.1. Data Unit

The atomic data element in Telco is a record which is presented in Figure 1.

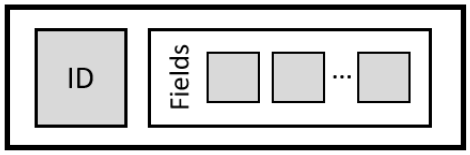


Figure 1 – Record representation

A record is an abstraction of a message and is made of 2 parts:

- **An identifier** - which relates the data to an entity. This entity may be a human user (Subscriber) or an equipment (Network Element);
- **A set of attributes (Fields)** - which characterizes the record. It holds data like timestamps, error/ success codes, geographical location, data volumes, etc.

Although multiple formats are used, data is commonly physical represented in simple and flat formats as sequence of bytes. Usually fixed separators divide fields, alternatively, records may follow a less optimized fixed length. Complex formats are also used, like the case of **Abstract Syntax Notation One** (ASN.1), with advanced and optimized serialization.

Most of Telco records follow a relatively short format, with sizes < 1 KB. The challenge is not record size, but total amount of records produced. A DSS system may easily have to handle > 10^6 records / daily. As a result, handling data for DSS Telco systems results in a big data exercise.

Records have different nature, which includes:

- **Subscriber Activities** – describe users' action in networks. May correspond to low level network signaling or with user activities already consolidated;
- **Network Element Indicators** – equipment generated indicators, more usually related with performance & resource consumption;
- **Network Element Alarms** – equipment alerts. Related to malfunctions or system degradation;
- **Characterization Data** – This set includes Customer Relationship management (CRM), equipment details (vendor, model and geographic location) or general business data.

3.2. Data Patterns

It is possible to identify the most frequent processing patterns applies to Telco records. The points below describe each pattern.

Aggregation

As presented before, Telco networks generates large amounts of data which introduces requirements for data reduction. Such techniques are not only necessary to process data in an acceptable time frame, but also to make it human possible to understand it.

Data aggregation pattern is presented in Figure 2.

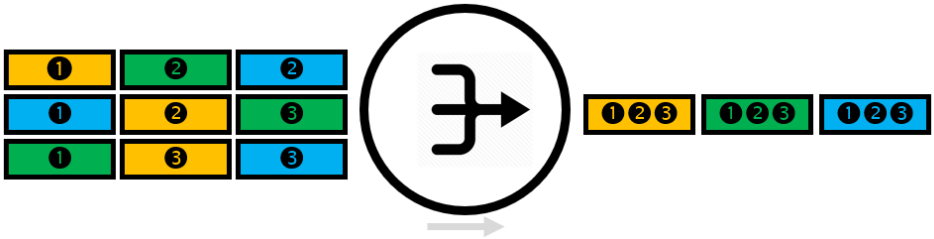


Figure 2 – Data aggregation pattern

As depicted in figure, reduction is realized by combining related records into a unique record. The pattern is divided into 2 flavors:

- **By Key** – Records with same key (*K*) are joined into a unique record (*U*). The produced record uses an aggregation function to combine the fields of each key ($V_{1 \dots n}$).

$$U \leftarrow Aggregation(K, \{V_1 \dots V_n\})$$

  Most common cases it to aggregate by Subscriber, Network Element or Geographical location. In this case, data reduction may decrease volumes between $[10;\ 10^3]$;

- **By Process** – A business logic is applied to a set of records (normally order in time) to produce a unique record. This case is common when receiving records from basic Telco procedures, like signaling and protocol workflows. In this case, data reduction may decrease between $[10;\ 10^2]$.

Aggregation pattern may require processing records ordered in time and handling temporary missing records. **caches** are often used to maintain **transient state**. If no order is required, the process is simplified, and an **incremental distributive function** is applied.

Correlation

When data from different record types needs to be merged together, pattern Correlation is used. The set of different records must have a **common key** to perform the match.

Data correlation pattern is presented in Figure 8.

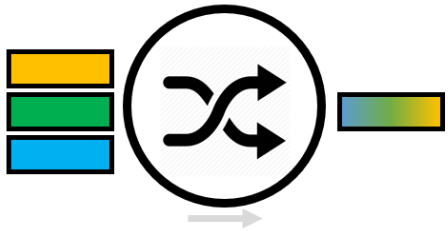


Figure 3 – Data correlation pattern

As depicted in figure, the result is a single record, with a composition of attributes from the source records. The complexity behind this pattern is that the new record may only be generated when all related data arrives. In most cases, there are no guarantees of message order. Such requirements force the introduction of **caches** to store short live data and definition of timeouts to clean data received but not totally matched.

Enrichment

It is possible to receive valid data records where some fields are not available. Such situation reflects Telco networks nature and its protocols.

Quite often the received records follow time-based workflows. In the first records, fields are provided that are not delivered in the following consequent records. Nevertheless, such fields are important for data processing and must be available in records.

Data segmentation pattern is presented in Figure 7.

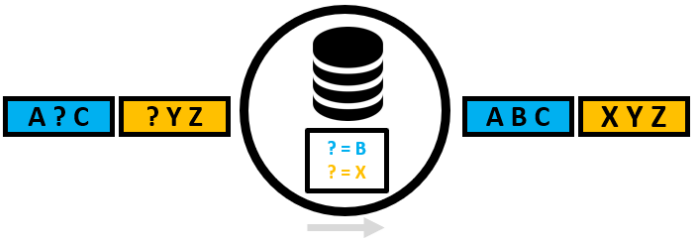


Figure 4 – Data enrichment pattern

As presented in picture, records are evaluated in search of missing fields. If a miss is detected, an internal **cache** is consulted in to verify the availability of the field. For performance and simplicity, the cache needs to implement a key-value data search pattern.

The cache is usually initialized with an initial set of data and updates its content as data arrives (learning). The update follows the concept of a **slow change dimension**, where the updates are < 1% than searches. If the update procedure becomes complex, the pattern leaves the scope of an enrichment and enters in the scope of a correlation.

Filtering

Often DSS have no control over data provided by data sources. Usually data sources follow an **all or nothing** concept, delivering to DSS all available information, even for not supported scenarios. It is up to DSS to eliminate unnecessary or redundant data.

Data filtering pattern is presented in Figure 5.

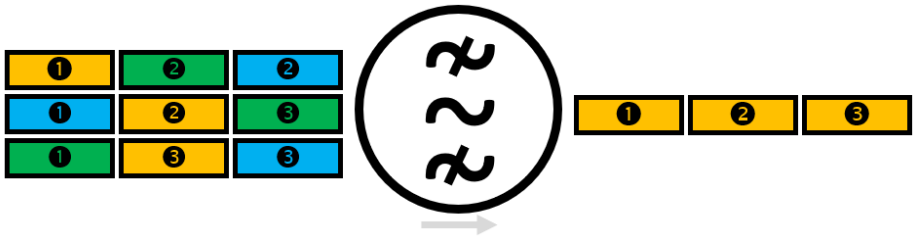


Figure 5 – Data filtering pattern

As presented in picture, a filtering function with a Predicate (*P*) is applied to every record.

$$Boolean \leftarrow Filtering(P)$$

If the predicate is true, the record is further processed, otherwise is eliminated from the data pipeline. Usually, eliminated records are diverted in the pipeline into logging infrastructures for counting and further evaluation.

In the end, Filtering is a practical way for introducing data reduction which may be applied in an early phase of the processing pipeline.

Replication

In replication a record is copied along parallel branches in data pipelines.

Data replication pattern is presented in Figure 6.

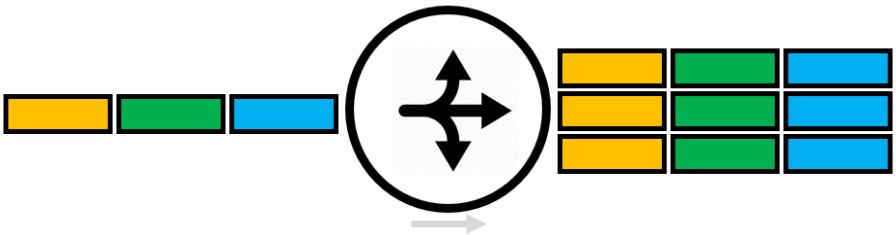


Figure 6 – Data replication pattern

As depicted in figure, the records are copied to multiple branches in the processing pipeline. The procedure implies a **deep copy** of the record content and, therefore, increase of system resources. As Telco records are usually flat, the copy procedure is a simple operation. The introduced penalty in resource consumption is compensated by parallel processing.

Segmentation

The large data amount produced by a Telco network is a big data challenge. To reduce its complexity magnitude, it is possible to use fundamental properties of its data and applied use cases. Many times, the necessary pipelines are oriented to entities processing (Subscriber, Network Elements or Services), where the computation of each entity is independent. For such cases, an early data segmentation allows a reduction in data complexity by using parallel resources.

Data segmentation pattern is presented in Figure 7.

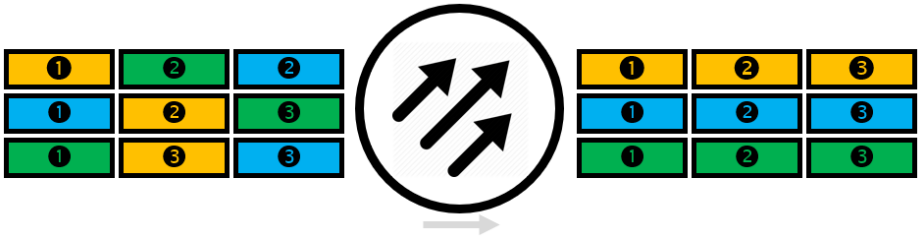


Figure 7 – Data segmentation pattern

As presented in picture, records are organized by a segmentation function able to group records by a characteristic (*C*) presented in data.

$$[1\,..n] \leftarrow Segmentation(C)$$

It does not matter if the characteristic is retrieved from a Key or Field. What matters is to have a Segmentation function that follows a **uniform distribution**, therefore providing a constant probability (as possible) for each produced element. This way the generated segments has a similar size.

Note that the processing of each segment is similar from a business logic point of view and are executed in parallel.

Validation

Records received by a DSS are often previously subjected to complex processing with multiple steps performed in different systems. As a result, **errors are expected** in the provided data sources. As a rule of thumb, an error rate ~ 1% is considered acceptable.

Errors are related to missing keys / mandatory fields, incorrect formats, invalid data types, out of bound data types and invalid enumerators field values. It is recommended to correct or eliminated such problems in an early phase of processing to avoid error propagation and producing incorrect insights.

Data validation pattern is presented in Figure 8.

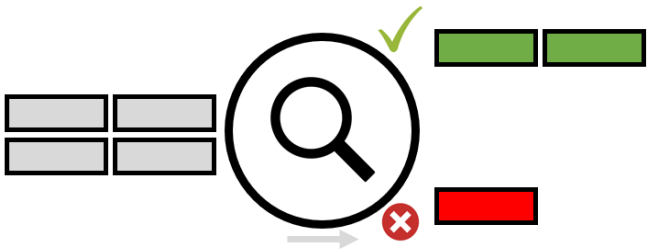


Figure 8 – Data validation pattern

As depicted in figure, validation checks each record verifying its content against the expected format. Valid records are considered for further processing. Invalid records may be rejected and eliminated from the processing pipeline or further processing for correction (if possible).

To build proper validation rules for a data source, it is necessary to receive formal documentation from data source providers. Such documentation must contain the data fields description, formats, mandatory/optional discriminators, sizes and enumerators lists.

Validation is also a form of data reduction, as invalid records are eliminated from further processing. Additionally, if executed in an early processing stage, it eliminates the need for further checks along the processing pipeline, contributing to faster and clear execution pipelines.

Note that validation use cases may be much more complicated. This is special true when validation requires analysing multiple related records. This means increasing the complexity of Validation per 1 additional data dimension.

4. Data Pipeline Scenarios

4.1. Introduction

This point focus in several data pipeline architecture principles which were common applied in the past but are now becoming outdated. For each point, the reason for its present limitations are presented. Additionally, a proved architecture alternative is presented.

Note that outdated architecture principles may be perfectly acceptable for specific solutions. It is up to the system designer to find the best practical solution, by evaluating the requirements (functional and non-functional. Over engineering a solution also leads to multiple issues. As a rule of thumb, follow the KISS principle.

4.2. Batch vs Realtime Processing

Constraint

Many data processing pipelines are centralized around batch processing. In such architecture, data arrives in **bulks**, typically provided as data files delivered at fixed periods. Processing is scheduled to run periodically and performs its actions **synchronously** and in **series**. Processing is divided into steps, working as intermediate checkpoints.

Issues

Bulks provide peaks in processor execution, while the remaining time, system is mostly idle. Bulk size and execution time window are extended in case of outage recover.

File usage introduces latency, as data is stored in a 3rd level memory system. Additionally, file accessing is most of the times sequential and single threaded.

High availability requirements force data to be kept in common storage systems, increasing system complexity and costs.

Solution

A solution to batch processing is to build **real-time processing** pipelines. Records are transformed into **messages** and processed as they arrive, avoiding latency and processing peaks. Software market offers

solutions where message delivery guarantee, routing and backpressure (steady data flow in case of bursts) are available as basic features. Processing is mainly executed at 2nd level memory. A real-time processing pipeline is also broken into multiple steps with checkpoints. However, the time between the steps are executed with minimal latency (aiming 0 latency).

As latency decreases, data is available to end users in a shorter time, a fundamental requirement for many systems.

4.3. Home Brew vs OEM

Constraint

It is common to have home brew components developed as data pipelines artefacts. Most of the times, this kind of solution is related with pressure to execute a costumer request and lack of knowledge of solutions already available in market. Created artefacts tend to stay and be adapted to other solutions, most times without proper internal restructuring or generalization.

Issues

Developing home based solutions are complex, expensive and require multiple resources. In the end, the solutions are specific to a costumer case and very difficult to reuse or generalize.

Besides the SW itself, it is necessary to design architectures, create documents and perform tests. A large investment and a considerable timeframe are necessary to provide product quality to n home brew component.

Solution

Software markets offers multiple products to efficiently implement data pipelines. Available components exist from **open source solutions** as Apache NIFI to **consolidated products** as Mediation Zone from Digital Route. The usage of such products contributes to faster integration cycles, more quality and focus on costumer cases.

4.4. Static vs Dynamic Documentation

Constraint

For understanding and maintenance, data pipelines need **documentation**. This documentation describes the operation steps and data transformations. Usually it is quite simple, provided in text and spreadsheet formats. It is manually created and updated.

Issues

Data pipelines have a complex development cycle, which is dependent on data quality and changes to source data. Additionally, in an initial deployment phase, changes are expected for correction and tuning of the pipeline. During all these phases, documentation is often neglected, becoming outdated. As documentation becomes outdated, its values decrease until a point it gets pointless. Without proper documentation, maintenance activities become hard and dangerous.

Solution

At least parts of the documentation may be generated automatic form source code and metadata. Software market offers different solutions from open source to consolidated products. An example is Doxygen. Such tools enable the **automation** of document creation, standardize formats, produce results in multiple formats and validate documentation metadata (forcing documentation as part of the pipeline creation process).

A special note for **visual process documentation**. Data pipeline can be presented visually, contributing for a better understanding of the complexity, dependencies and business logic. In fact, in following points of this document, a proposed visual documentation is used.

4.5. Horizontal vs Vertical Scaling

Constraint

Traditional data pipeline solutions are created in single server (or a small number of servers, providing high-availability). As performance and data volumes requirements increase, **vertical scaling** (increase in processor and memory systems) is the most common answer.

Issues

Although server processing and storage has increased in last years, data volumes have increase even more. This means that for many cases, horizontal scaling is not enough to cope with processing requirements.

Architectures with single server (or a short number of servers) are also subjected to HW constraints. So, it may happen that a server is subjected to a vertical scaling, but system limitations are related to underlying infrastructure like memory systems or network access.

High-availability scenarios are also complicated. It forces the use of redundant components, many times in stand-by mode and therefore not contributing for system processing.

Solution

Horizontal scaling (adding more servers) may be the only solution for big and fast data use cases. Most of the technology for managing a cluster of servers is now simplified, by the introduction of technologies like Apache Hadoop and its related projects.

This architecture is further enhanced by out-of-the box high-availably. Additionally, many use cases in Telco domain are **purely distributed** and may be executed in independent servers as no shared data is required or dependable within the use case. So, data may be routed to independent servers without the need of extended shuffling between servers.

4.6. Static vs Reactive Applications

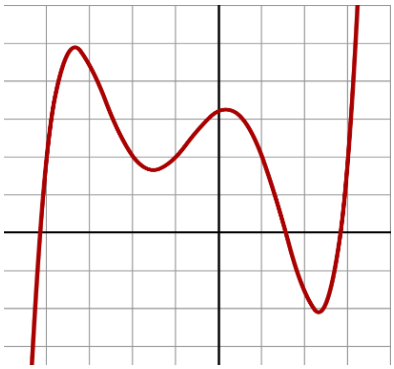
Constraint

Artefacts in a data processing pipeline are applications running over a platform (Operating System). Traditionally, applications have a **static behaviour** with limited flexibility on how resources provided by the underling platform are used. In static applications, resources are managed (mostly) by basic configuration.

Issues

Resource usage flexibly is only provided by means of configuration parameters passed during system setup or complex operations executed by system administrators. Most times, changing such parameters require a system restart to take effect.

In telco domain, daily data volume received usually follows a **Quintin Function** (polynomial of degree 5). The function is presented in Figure 9.



$$g(x) \leftarrow ax^5 + bx^4 + cx^3 + dx^2 + ex + k$$

Figure 9 – Quintin function

As presented in the figure, daily traffic has 2 peaks. The 1st peak happens before lunch time, the 2nd at late afternoon. During the night, data decreases significantly. Note that this curve flows the expected behaviour for normal working days. Weekends and holidays may produce different results. As a result, from received data volumes, it makes sense for applications to have an adaptative behaviour in time. Resources may be released during the night and allocated during rush hour.

Solution

A solution to static applications is reactive design. The concept was initially introduced by the **Reactive Manifesto**. The manifest claims for applications which are more flexible, loosely-coupled and scalable, by promoting 4 architecture principles:

- **Responsive** - responses in a timely manner as possible;
- **Resilient** - stay responsive in the case of failure;
- **Elastic** - adaptable workload;
- **Message Driven** - use asynchronous messages to establish a boundary between components, ensuring loose coupling, isolation and location transparency (independent service location).

4.7. Monoliths vs Microservices

Constraint

Traditionally, DSS systems are implemented in **monolithic client-server** architectures. Quite often the architecture has a 3-level tier with a rich client (more recently web based), a server with a public API (implementing all business logic) and a storage system supported by a commercial database.

Issues

In case of issues with non-functional requirements, the only practical way to address the problem is using horizontal scaling (with the problems presented above).

Maintenance is also an issue. As new features are introduced, complexity in development and testing increases.

Solution

A solution is to use **microservices**. With microservices, business logic is divided into smaller and more manageable components. Such components may be distributed in a cluster and managed according to demand, therefore following the reactive principles presented above.

Software markets simplifies the creation and deployment of microservices by providing frameworks as Lagom from Lightbend.

5. Architecture

5.1. Introduction

This point presents an architecture for a data processing pipeline supporting big and fast data. The architecture implements the data patterns from point 3.2 - Data Patterns and the scenarios from point 4 - Data Pipeline Scenarios.

The architecture description starts by presenting the logical architecture with the definition and requirements of each component. It is followed by the physical architecture with a proposal for its actual instantiation.

5.2. Logical Architecture

The logical architecture is presented in Figure 10.

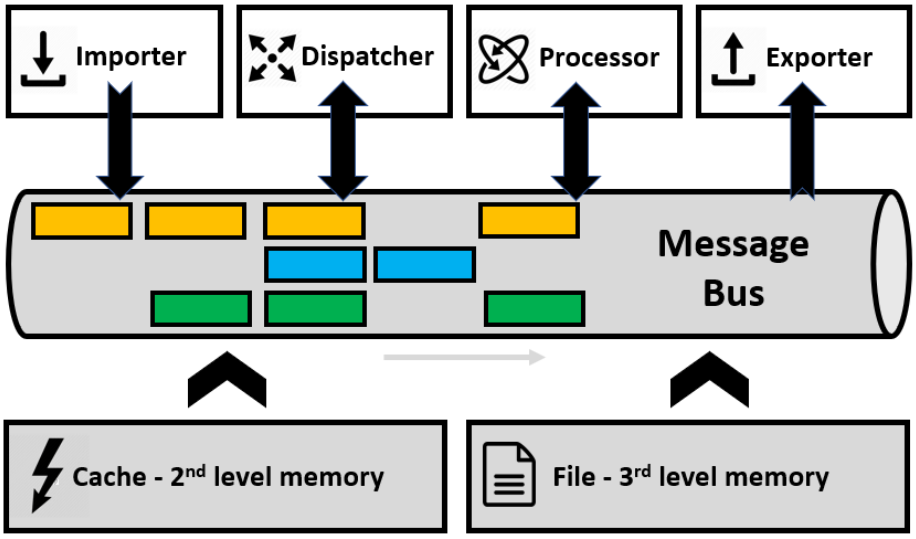


Figure 10 – Logical Architecture

As presented in the figure, there is a clear distinction between components which are consider as infrastructures (greyed out) and business logic components.

The infrastructure components are the system capabilities provided out of the box. Business logic components require a specific implementation, but its flexibility/reusability may be quite high, and its development based on the infrastructure and other 3rd party artefacts.

The architecture backbone is based on a Message Bus. This Bus allows the distribution of asynchronous messages. Business logic components read and write over the Bus, enriching the Bus functionality. The Bus provides multiple logical independent data channels for parallel processing.

Cache

A cache is a **fast data access** component infrastructure, implemented over 2nd level memory systems. It provides a **low latency** for **transient data** storage. Caches are quite useful during algorithm stages as **data checkpoints**. To understand the role of a cache, it is important to note that the difference between 2nd level and 3rd level memory access is usually in orders of magnitudes from nanoseconds to milliseconds.

This component is part of the underlying infrastructure.

Dispatcher

The dispatcher has the capability to **read**, **transform** and **write** messages in the bus. This means the component uses the Application Programming Interface (API) provided by the message bus for interaction. As a principle, the component does not interact with other components. After applying its business logic, data is written back to the bus, eventually to new logical channel(s).

The level of generalization of a Dispatcher component may be high. Configuration data may be used to increase the component reusability.

This component implements the data patterns of Filtering, Replication, Segmentation and Validation. It may also be used for simplifications of pattern Aggregation, Enrichment and Correlation.

Exporter

The exporter has the capability to **transfer** messages from the bus to other devices. It transforms the read messages into data structures which are transferred to other systems. Typical cases are databases (SQL and NO-SQL), file systems, analytics tools or specific 3rd party systems.

Exporters perform basic data conversions, although more complex transformations are expected to be executed with the Dispatcher component. Data formats transformations are expected, as internal bus message use more condensed formats (for machine performance) and output formats more extended ones (for human readability).

Careful must be taken while writing Exporters, due to the latency of the destination systems. To minimize this problem, the Exporters may rely on the Cache for buffering data.

File System

File system represents the 3rd memory level, which offers a **permanent store** media. Although subject to higher latency than other memory level systems, it is fundamental to store long term configuration and processed data.

This component is part of the underlying infrastructure.

Importer

The importer has the capability to **feed** messages into the Bus. It transforms data from home brew and 3rd party protocols/formats into single messages.

Importers are expected to perform data validation, selection and conversions. The usage of basic data types and shorter messages contribute to an overall performance enhancement with the message bus.

This component implements the data patterns of Filtering and Validation. It may also be used for simplifications of pattern Enrichment.

Message Bus

As presented before, the message bus is the backbone of the architecture. It offers an infrastructure for dispatching **asynchronous messages** and an API for reading and writing to the Bus. The message concept fits perfectly in the definition of a Telco record presented in point 3.1 - Data Unit.

Messages are handled in **virtual channels**. Each channel is an independent virtual path, operated in parallel. Channels have independent properties for advance tuning, according to different types of messaging requirements.

The bus relies on the 2nd or 3rd memory level for persistence and message delivery assurance. Message acknowledge, replication and fault tolerance are mandatory features. The bus supports fast encoders/decoders for the typical data structure of Telco records.

This component is part of the underlying infrastructure.

Processor

The processor has the capability to implement **complex business logic**. It supports the concept of message windowing and state persistence between message arrival.

Message window corresponds to the aggregation of related messages into micro batches. At a regular interval size, it is possible to consolidate all receive messages in a single processing block. At the same time, the processing from other previous intervals is also available as stateful information. This stateful information can be used as lookup data (useful for Enrichment and Correlation) or as processing stage (applicable for Aggregation).

For performance reasons, it is advised for state data to be stored in Cache component as an alternative to File System component. This is only possible for transient data, but the stages of an aggregation process are transient until the final state is achieved.

This component implements the data patterns of Aggregation, Enrichment and Correlation.

5.3. Physical Architecture

The physical architecture is presented in Figure 11.

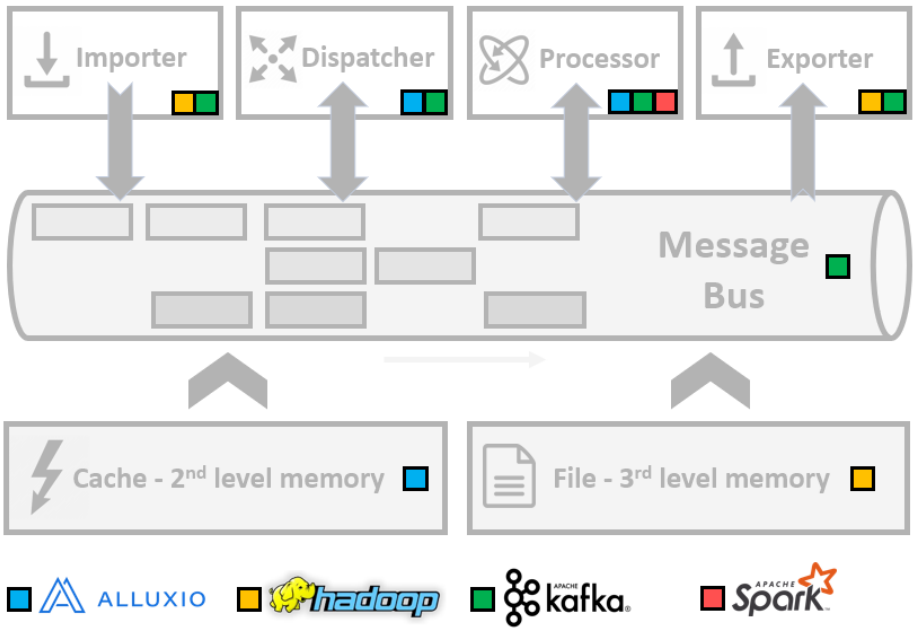


Figure 11 – Physical Architecture

As presented in the figure, each logical component is assigned with a set of SW market technologies as its implementation support. The chosen technologies follow the principles presented in point 4 - Data Pipeline Scenarios. All chosen technologies support cluster computing and provide out of the box the features for high-availability, security, data distribution and parallel data processing.

NOTE: The following points present the technologies and its usage within the architecture. It is considered that the reader is familiar with the presented technologies, as so, no description of its and concepts is presented.

Alluxio

Alluxio is an open source **virtual distributed storage**. It enables an application to interact with storage data at memory speed time. It also provides an abstraction layer for different permanent storage system. In fact, Alluxio is a distributed 2nd level memory system.

The features provided by Alluxio cover the requirements for the infrastructure Cache component with the additional capability of cluster distribution with data replication. Two important features are out-of-box integrations with Spark and the Hadoop File System.

Alluxio is available for the Dispatcher and Processor components to access and store transient lookup data and state. Its low latency contributes as an important performance booster.

Hadoop (File System)

Hadoop Distributed File System (HDFS) is the **primary data storage** system used by Hadoop applications. It implements a distributed file system providing high-performance access to data across Hadoop clusters. In fact, HDFS is a distributed 3rd level memory system.

It is important to note that the Hadoop ecosystem provides a large set of tools, ranging from resource managers, to databases and machine-learning platforms. All these related projects are integrated and use the common components of Hadoop.

Being a permanent storage, HDFS is available for the Importer and Exporter component as data source and destination, respectively. Care must be taken while interacting with this component due to its latency.

Kafka

Kafka is a **distributed streaming** platform. It processes asynchronous messages in real-time, providing mechanisms for publishing and consuming those messages in a cluster with fault-tolerance, ordering and guarantee deliver.

The main features of Kafka fit into the **Message Bus** component requirements. Kafka messages are organized into topics providing virtual parallel channel processing. Topics are further divided into partitions for distribution and performance. The basic structure of a Kafka message (called a Kafka record) is a key and a value, which fits to the basic data unit presented in Figure 1. Kafka advises

compact data formats for fast encoding/decoding and the usage of performant serialization frameworks live Avro. This is a perfect match for the for the simple and shallow records required in Telco data units.

An important feature of Kafka is its set of public APIs to write data importers and exporters, respectively called Producers and Consumers. Additionally, the APIs allow to write and manage pure streaming business logic (Kafka Connectors and Kafka Streams). The APIs area available for programming with languages Java and Scala.

Commercial platforms like Confluent extend the basic functionality of Kafka, providing advanced features for deploying, monitor and management. The return of investment in such platform is high as it simplifies operation and alerts to potential bottlenecks.

Kafka and its APIs are the underlying platform for Importer, Dispatcher and Exporter components. It also serves as a data source for the Processor component.

Spark

Spark is a fast and general-purpose **cluster computing** system. One of its main features is Spark Streaming, an extension to the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.

With Spark streaming it is possible to define processing windows that work as micro-batches. Such windows receive direct messages from Kafka as Discretized Stream (DStream). A DStream represents a continuous stream of received input data. Internally, a DStream is built by a continuous series of RDDs (Resilient Distributed Datasets), which is Spark's abstraction for distributed immutability. Each RDD in a DStream contains data from a certain interval and is instantiated through the cluster.

Besides streaming, Spark provides a set of APIs for distributed computing, SQL representation, cluster graphs (GraphX) and machine learning (MlLib).

Spark Streaming is the underlying platform for the Processor component.

Other Technologies

Although not consider in the physical architecture, the following technologies may also be considered as part of the solution:

- Yarn - Used to arbitrate the resources available in a Hadoop cluster. Necessary for managing resources delivered to Spark applications;
- Akka - Processing framework based on the Actor model. It is a possible solution for developing Processor components with complex business logic;
- Lagom - Already referenced above, it is a framework for developing and managing microservices;
- Flink - framework and distributed processing engine for stateful computations with streams.

6. Example

Figure 12 in point 8 - Appendix A – Data Pipeline exemplepresents an example for a data pipeline built with the architecture developed in this document. Each step is represented with a box divided into 2 parts. The top part shows the chosen **component**, while the bottom its implemented **patterns**. The pipeline flows from left to right. Each element is identified with a step ID, described below.

Step 1

There are 2 data sources (yellow and blue) provided as input data. To load the data sources into the pipeline, **Importer** components are used. Besides data import, these stages also implement the patterns of **Validation** and **Filtering**. In case of errors detected during Validation, records are sent to Stage 2, for logging. Otherwise data is pushed into the pipeline for further processing.

From a physical point of view, this component is implemented with **Kafka** API.

Step 2

Incorrect records are retrieved from the pipeline and stored in the 3rd level memory system. An **Exporter** component is used to store the records.

From a physical point of view, this component is implemented with **Kafka** API and **3rd level memory system**.

Step 3

Business logic for the yellow interface require a simple enrichment with basic lookup data. A **Distributor** component is used, implementing the **Enrichment** pattern.

From a physical point of view, this component is implemented with **Kafka** API and **2nd or 3rd level memory system**.

Step 4

The yellow and blue data sources are correlated to generate the green data source. A **Processor** component is used, because the business logic associated with the **Correlation** pattern is complex.

From a physical point of view, this component is implemented with **Kafka** API, **Spark Streaming** and the **2nd level memory system**.

Step 5

The green data source is replicated to different parallel channels. A **Distributor** component is used with a **Replication** pattern. Data transformation necessary at this step is minimal. Basic logic for the distribution applies the usage of Filtering pattern.

From a physical point of view, this component is implemented with **Kafka** API.

Step 6

Each parallel pipeline created in the previous path allows the creation of different parallel aggregation processors components. The aggregations may be distinct in the used aggregation keys, time frames and business logic. Typical aggregations have a time based, which may be different per channel. After the aggregation window is closed, data is written back to the Bus in a specific path. For this process, a **Processor** component is used, and the **Aggregation** pattern implemented.

From a physical point of view, this component is implemented with **Kafka** API, **Spark Streaming** and the **2nd level memory system**.

Step 7

As the last step, data is written by **Exporter** components to a **3rd level memory system** to guarantee permanent storage.

From a physical point of view, this component is implemented with **Kafka** API, **Spark Streaming** and the **3rd level memory system**.

7. Conclusion

As costumer requirements evolve to larger data sets and close to real-time processing, DSS systems must be able to provide solutions in the realm of **big and fast data**. This document describes a possible approach, presenting basic data patterns, a visual representation proposal, logical and physical architecture.

As presented in the document, the patterns and logical concepts are **available in the market**, materialized in stable and performant products. It is a matter of following the **best guidelines**, apply the recommended **concept & tools** and be **bold**.

From these ideas, it is necessary to find human and material resources and execute an exploratory program to prove and adapt the concept. The result of such work is a **blueprint** to be used for building the next generation data pipelines.

8. Appendix A – Data Pipeline exemple

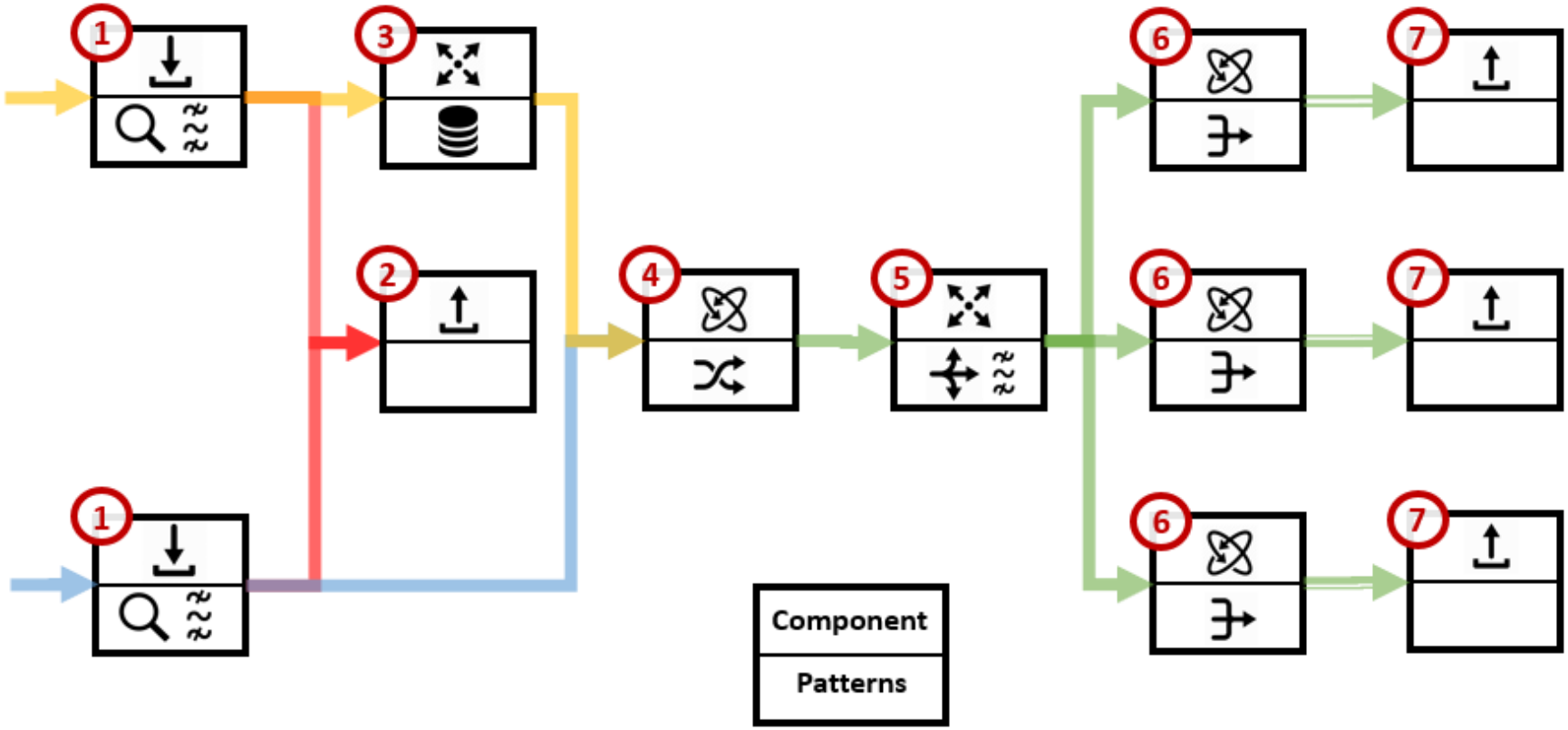


Figure 12 – Data pipeline example