Universidad Torcuato Di Tella
Tecnología Digital

Trabajo Práctico 2

TDVII - Ingeniería de Datos

Arquitectura y flujos de datos

Integrantes:

JAHDE Josefina
JENKINS Dafydd
CASTORE Juan Ignacio
CHEN Belén Débora

Semestre 1 2025

Introducción

En la primera parte de este trabajo, nos enfocamos en modelar una base de datos relacional orientada a relevar y analizar los indicadores de salud en los efectores de salud de las provincias argentinas. Esto incluye aspectos como la población registrada, los efectores de salud, los tipos de problemas con los que lidian, la implementación de Historias Clínicas Electrónicas (HCE) y la capacidad de emitir recetas digitales, entre otros puntos importantes.

Ahora, en esta segunda entrega, nos encargamos del diseño de una arquitectura de datos que respalda el modelo conceptual y lógico que ya definimos, y además incluye un sistema de persistencia políglota que se adapte a los diferentes tipos de datos y necesidades de procesamiento. Para lograr esto, definimos arquitecturas, flujos de datos, procesos de validación y diversas soluciones tecnológicas que enriquecen y aseguran la calidad de la información en cada etapa.

Este informe detalla el desarrollo de consultas utilizando Spark SQL y MapReduce, el uso de reglas de validación con Great Expectations, el uso de Redis para la gestión de datos clave-valor, y el diseño de estructuras en MongoDB para representar documentos relevantes al ámbito de la salud.

Arquitectura

En esta sección se detalla la arquitectura de datos propuesta para el sistema de indicadores de tecnología en salud, considerando su implementación en el contexto de una organización pública real, como un Ministerio de Salud provincial o nacional. Dado el volumen, variedad y complejidad de los datos involucrados, se adopta un enfoque basado en almacenamiento distribuido y procesamiento mixto batch y en tiempo real, enmarcado dentro del patrón arquitectónico conocido como **arquitectura Lambda**.

a) Origen, formato, cadencia y volumen de los datos

Los datos provienen de diversas fuentes institucionales dentro del sistema público de salud. Por un lado, los *efectores de salud* (hospitales, centros de atención primaria, instituciones especializadas) reportan información sobre su infraestructura, servicios brindados y grado de implementación tecnológica. Por otro lado, las áreas de gestión de personas registran datos sobre la población federada, sus coberturas médicas y su relación con el sistema de salud.

Estos datos son generados de manera distribuida por cada jurisdicción y arriban al sistema mediante archivos en formato `CSV` o `JSON`, o a través de integraciones con APIs. La frecuencia de actualización varía según la fuente: los datos de personas federadas pueden registrarse a diario, los datos de efectores suelen actualizarse mensualmente (salvo quizás la relacionada a sus problemas, que podría actualizarse diariamente), y la información relativa a sistemas de Historia Clínica Electrónica (HCE) se modifica sólo cuando hay cambios en la infraestructura tecnológica.

Dada la escala del sistema (a nivel nacional), se estima que el volumen de datos es considerable: cientos de miles de personas por provincias, por 23 provincias dan un total de millones de registros de personas. Miles de efectores por provincia por 23 provincias dan un total de decenas de miles de efectores, con sus respectivas decenas de atributos técnicos.

Esto justifica la adopción de una solución de almacenamiento distribuido como **HDFS**, **Amazon S3** o **Delta Lake**, capaz de soportar crecimiento a gran escala y consultas intensivas.

b) Linaje de los datos y procesos intermedios

La arquitectura Lambda contempla dos flujos principales: el **batch**, que procesa grandes volúmenes de datos de forma periódica, y el **streaming**, que permite manejar información en tiempo real o con baja latencia.

En el flujo batch, los datos llegan en crudo a una *zona de aterrizaje* (landing zone) dentro del sistema distribuido. Allí se almacenan sin transformar, permitiendo trazabilidad completa y posibilidad de reprocesamiento. A continuación, se ejecutan procesos ETL desarrollados en **Apache Spark**, que incluyen validaciones de calidad con **Great Expectations**, limpieza, transformación, enriquecimiento semántico (por ejemplo, mapeo a SNOMED CT) y estandarización de estructuras. Los datos procesados se almacenan nuevamente en el sistema distribuido, en formatos optimizados como `Parquet` o `Avro`.

Simultáneamente, el flujo en tiempo real está soportado por **Redis**, donde se mantienen claves temporales (como tareas urgentes, alertas operativas o buffers de ingestión rápida). Esta capa permite respuestas inmediatas ante eventos críticos, sin depender del ciclo batch completo.

De esta forma, se garantiza tanto la integridad y calidad de los datos históricos como la agilidad para operar con datos recientes.

c) Usos de los datos en las distintas etapas

Durante la ingestión, los datos son utilizados por los equipos técnicos para verificar consistencia y registrar metadatos de carga. En la fase de procesamiento, se construyen datasets confiables que alimentan sistemas de monitoreo, paneles de control y análisis estadísticos.

En el almacenamiento principal (podría ser un *data warehouse*), los analistas pueden ejecutar consultas complejas para obtener indicadores sanitarios y tecnológicos, tales como la proporción de efectores sin problemas de conectividad o la evolución de la cobertura pública exclusiva (CPE) en cada provincia.

En la capa real-time, Redis y Spark permite gestionar tareas operativas como validaciones pendientes, seguimiento de carga o detección de valores anómalos con TTL (tiempo de vida limitado).

Finalmente, los datos consolidados se disponibilizan a sistemas de visualización, reportes institucionales y APIs internas que permiten el acceso controlado a otras áreas del Estado.

d) Roles y permisos de uso

El sistema será utilizado por múltiples actores institucionales con distintos niveles de responsabilidad y acceso:

- **Administradores nacionales**: acceso total a los datos, responsables del mantenimiento general, monitoreo y auditoría del sistema.
- **Técnicos provinciales**: carga y gestión de datos propios de su jurisdicción, incluyendo personas, efectores y sistemas HCE.

- **Analistas de modernización**: acceso de lectura a indicadores consolidados para evaluar el grado de avance tecnológico y priorizar intervenciones.
- **Usuarios operativos**: carga acotada a determinados formularios o efectores bajo su responsabilidad.
- **Auditores**: acceso de solo lectura sobre datos validados para fines de seguimiento institucional.

Esta segmentación de permisos garantiza seguridad, trazabilidad y eficiencia operativa, permitiendo a cada usuario trabajar sobre su dominio de datos sin interferir con otros niveles del sistema.

Map Reduce con Spark

En esta sección, se presentan consultas resueltas utilizando el paradigma MapReduce con Apache Spark, detallando las fases de Map, Reduce y Recolección de datos para cada una. Las consultas que decidimos mostrar se realizan sobre los datos de efectores de salud, provincias y personas.

1. Cantidad de Efectores de Salud por Provincia

Esta consulta tiene como objetivo calcular el número de efectores de salud presentes en cada provincia.

- **Fase de Map:** Se toma el RDD 'efectoresdesalud', que contiene tuplas '(cod_refes, nombre, id_prov, nivel)'. La función de mapeo transforma cada elemento en un par '(id_prov, 1)'. Esto asigna un contador de 1 a cada efector de salud, agrupado por su 'id_prov'. Además, se mapea el RDD 'provincias' a '(id_prov, nombre)' para obtener los nombres de las provincias.
- **Fase de Reduce:** Se aplica 'reduceByKey' sobre el RDD resultante de la fase de map ('efectores_por_prov'). La función de reducción suma los contadores para cada 'id_prov', resultando en '(id_prov, cantidad de efectores)'. Luego, se realiza un 'join' con 'prov_nombre' para asociar la cantidad de efectores con el nombre de la provincia, y así sea más fácil de leer.
- **Fase de Merge:** El resultado final se obtiene mediante un 'map' que reordena los elementos a '(nombre, cantidad de efectores)' y se recolecta utilizando 'collect()' para imprimir los pares '(provincia, cantidad_efectores)'.

2. Promedio de Edad por Provincia

Esta consulta calcula la edad promedio de las personas registradas en el sistema de salud por cada provincia.

- **Fase de Map:** Se toma el RDD 'personas', que contiene tuplas '(id_provincia, fecha_nacimiento)'. La función de mapeo calcula la edad de cada persona y emite un par '(id_provincia, (edad, 1))'. El 1 se usa luego como contador para el promedio.

- **Fase de Reduce:** Se aplica 'reduceByKey' sobre el RDD 'edades'. La función de reducción suma las edades y los contadores '(suma_edades, suma_contadores)' para cada 'id_provincia'. Luego, se calcula el promedio de edad para cada provincia usando 'mapValues' y se realiza un 'join' con el RDD de provincias para obtener los nombres.
- **Fase de Merge:** El resultado final se recolecta con 'collect()' y se imprime el nombre de la provincia y su edad promedio.

3. Cantidad de HCE que gestionan Recetas Digitales

Esta consulta determina cuántas Historias Clínicas Electrónicas (HCE) tienen la capacidad de generar recetas digitales.

- **Fase de Map:** Se toma el RDD 'hce', que contiene el estado de 'generacion_receta_digital' ('Genera' o 'No genera') para cada HCE. La función de mapeo transforma cada estado en un par '(estado, 1)'.
- **Fase de Reduce:** Se aplica 'reduceByKey' sobre el RDD 'mapeo'. La función de reducción suma los contadores para cada estado de generación de receta digital, resultando en '(estado, cantidad)'.
- **Fase de Merge:** El resultado final se recolecta con 'collect()' y se imprime cada par '(estado, cantidad)'.

4. Proporción de Personas Registradas en el Sistema de Salud por Provincia

Esta consulta calcula la proporción de personas registradas en el sistema de salud respecto a la población total de cada provincia.

- **Fase de Map:** Se utilizan dos RDDs: 'personas_rdd' (con '(id_provincia, fecha_nacimiento)') y 'prov_rdd' (con '(id_prov, nombre, poblacion_total)').
  - Para 'personas_rdd', la función de mapeo genera '(id_prov, 1)' para contar las personas registradas por provincia.
  - Para 'prov_rdd', se generan dos RDDs mapeados: '(id_prov, poblacion_total)' y '(id_prov, nombre)'.
- **Fase de Reduce:** Se aplica 'reduceByKey' sobre el RDD 'registradas' para obtener la cantidad de personas registradas por provincia '(id_prov, cantidad de personas)'. Luego, se realiza un 'join' entre 'prov_poblacion_total' y el 'conteo' de personas registradas para obtener '(id_prov, (poblacion_total, cantidad de personas))'. Finalmente, se calcula la proporción utilizando 'mapValues' y se une con los nombres de las provincias.
- **Fase de Merge:** El resultado final se recolecta con 'collect()' y se imprime el nombre de la provincia y la proporción calculada.

Spark SQL

En esta sección, se presentan consultas realizadas utilizando Spark SQL sobre los DataFrames cargados a partir de archivos CSV. Los resultados se muestran en formato de tabla. Se utilizaron las tablas 'problema', 'efectordesalud', y 'provincia'.

Carga de Datos y Creación de Vistas Temporales

Se cargaron los siguientes datasets en DataFrames de Spark y se crearon vistas temporales para poder ejecutar las consultas SQL directamente:

- 'problema.csv' como tabla 'problema'.
- 'efectordesalud.csv' como tabla 'efectordesalud'.
- 'provincia.csv' como tabla 'provincia'.

1. Cantidad de efectores de salud que tienen problemas por nivel.

Esta consulta identifica la cantidad de efectores de salud distintos que reportan problemas, agrupados por su nivel (Primer Nivel, Segundo Nivel, Tercer Nivel).

```
SELECT e.nivel
    , COUNT(DISTINCT e.cod_refes) as cant_efectores_con_problemas
FROM efectordesalud e
    JOIN problema p ON e.cod_refes = p.cod_refes
GROUP BY e.nivel
ORDER BY cant_efectores_con_problemas DESC
```

2. Cantidad de problemas de cada tipo por provincia.

Esta consulta muestra el recuento de cada tipo de problema ('Infraestructura', 'Conectividad', 'FaltaDispositivos') en cada provincia.

```
SELECT
  pr.nombre provincia
  , p.tipo
  , COUNT(*) cantidad
FROM problema p
  JOIN efectordesalud e ON p.cod_refes = e.cod_refes
    JOIN provincia pr ON e.id_prov = pr.id_prov
GROUP BY pr.nombre, p.tipo
ORDER BY provincia, cantidad DESC
```

3. Cantidad de problemas por provincia, por habitante.

Esta consulta calcula la proporción de problemas reportados por habitante para cada provincia, lo que permite identificar la densidad de problemas en relación con la población.

```
SELECT pr.nombre provincia,
      COUNT(p.cod_problema) cantidad_problemas,
      pr.poblacion_total,
      (ROUND(COUNT(p.cod_problema) / pr.poblacion_total, 6)) problemas_por_habitante
FROM problema p
    JOIN efectordesalud e ON p.cod_refes = e.cod_refes
        JOIN provincia pr ON e.id_prov = pr.id_prov
GROUP BY pr.nombre, pr.poblacion_total
ORDER BY problemas_por_habitante DESC
```

Great Expectations

Para asegurar la calidad de los datos procesados en este sistema, se pueden aplicar validaciones automáticas con la herramienta **Great Expectations**, que permite detectar errores de estructura, tipo o contenido antes de ejecutar transformaciones complejas.

A continuación se describen cinco validaciones posibles, inspiradas en patrones comunes de control de calidad:

1. **Tipo de datos en el campo de población**
   En el dataset de provincias, se puede validar que la columna `poblacion_total` sea del tipo `INTEGER`:

   ```
   expect_column_values_to_be_of_type("poblacion_total", "INTEGER")
   ```

2. **Formato del nombre de efectores con expresión regular**
   En el archivo `efectordesalud.csv`, se puede aplicar una expresión regular para validar que los nombres de efectores comiencen con palabras institucionales estándar como `Hospital`, `Centro`, `Clínica`, etc.:

   ```
   expect_column_values_to_match_regex(
       "nombre",
       r"^(Hospital|Centro|Clínica|Posta|Instituto|Unidad)"
   )
   ```

3. **Dominio cerrado en la columna `nivel`**
   Se puede controlar que la columna `nivel` de los efectores contenga sólo valores válidos: `"Primer Nivel"`, `"Segundo Nivel"` y `"Tercer Nivel"`:

   ```
   niveles = {"Primer Nivel", "Segundo Nivel", "Tercer Nivel"}
   expect_column_values_to_be_in_set("nivel", niveles)
   ```

4. **Validación de unicidad compuesta**
   Para `EfectorDeSalud`, se puede validar que la combinación de columnas `id_prov` y `nombre` sea única, para evitar duplicados en registros clave:

```
expect_compound_columns_to_be_unique(["id_prov", "nombre"])
```

5. **Rango válido de población**
   Finalmente, se puede validar que los valores de `poblacion_total` en el archivo `provincia.csv` estén entre 10.000 y 20.000.000 habitantes:

```
expect_column_values_to_be_between(
    "poblacion_total", 10000, 20000000
)
```

Estas validaciones ayudan a prevenir inconsistencias o errores antes de procesar los datos con herramientas como Spark o cargarlos a bases no relacionales como MongoDB.

Redis

En esta sección se utiliza **Redis** como base clave-valor para almacenar datos del dominio de salud en estructuras eficientes como strings, listas y hashes. Se trabajó sobre tres aspectos principales: almacenamiento de datos individuales, gestión de listas con operaciones básicas, y uso de claves con expiración temporal (TTL).

1. Almacenamiento clave-valor de entidades individuales

Se simularon registros de personas (por ejemplo, datos de una paciente como nombre, apellido, documento, cobertura médica, etc.) y se almacenaron como strings en Redis usando el modelo clave-valor. Se usó una clave única para cada persona (`persona:1000`) y se serializó el diccionario correspondiente en formato JSON para facilitar su almacenamiento.

En primer lugar, lo cargamos manualmente, definiendo el diccionario con los datos de la persona nosotros. Almacenamos a la persona como (clave, json).

Luego, la información fue recuperada, modificada (por ejemplo, cambiando la cobertura médica) y actualizada nuevamente en Redis, utilizando los métodos `set()` y `get()`.

Después, mostramos como sería cargar los datos desde un json almacenado en un repositorio en GitHub (generado a partir de los datos del TP1) y almacenando todas las personas como (clave, json) en un Hash con nombre "personas".

Por último, mostramos como se eliminarían o agregarían nuevas personas manualmente a dicho Hash.

```
r.set("persona:1000", json.dumps(persona_dict))
lidia = json.loads(r.get("persona:1000"))
lidia["cobertura_medica"] = "CPE"
r.set("persona:1000", json.dumps(lidia))
```

2. Lista de efectores que no generan recetas digitales

A partir los datos del primer TP, se extrajo una lista de efectores que no poseen generación de recetas digitales. Estos valores fueron almacenados como una lista en Redis bajo la clave `"hce_no_genera_recetas"`, utilizando el método `rpush()` para cargar todos los elementos. La idea es utilizar la estructura de lista como una "Lista de HCEs por modernizar"(como una especie de lista de tareas).

Posteriormente se demostró la recuperación de dicha lista mediante `lrange()`.

```
r.rpush("hce_no_genera_recetas", *hce_no_gestiona)
print(r.lrange("hce_no_genera_recetas", 0, -1))
```

Este tipo de estructura resulta útil para gestionar listas de tareas pendientes.

3. Uso de TTL en claves de problemas urgentes

Redis permite definir un tiempo de expiración para cada clave, lo que resulta útil para modelar información transitoria. En este trabajo, se representaron problemas críticos reportados por efectores de salud como objetos JSON y se almacenaron con claves como `problema:100`.

Para simular urgencias, se usaron distintos TTLs (Time-To-Live): uno de 24 horas, otro de 2 horas y otro de apenas 10 segundos. Por ejemplo:

```
r.setex("problema:102", 10, json.dumps(problema3))
```

Además, se implementó un bucle que monitorea estos problemas y muestra un mensaje si:

- Queda menos de una hora para que expire un problema (se lo considera urgente).
- El problema ha expirado (ya no está en Redis).

Esto permite gestionar alertas automáticas sobre eventos críticos con duración limitada en el tiempo. Redis actúa así como un buffer temporal de eventos urgentes, útil para flujos operativos en tiempo real.

MongoDB

En esta sección, se presentan consultas resueltas utilizando MongoDB. Se trabajó sobre una base de datos denominada `indicadores-de-salud`, en la cual se cargaron automáticamente archivos JSON desde un repositorio de GitHub, creando colecciones como: `datos-efectordesalud`, `datos-hce`, `datos-persona`, `datos-provincia`, `datos-problema`, entre otras.

A partir de esta estructura, se desarrollaron distintas consultas utilizando `pipelines de agregación`. Las consultas realizadas fueron:

1. **Listar todas las personas con Cobertura Pública Exclusiva (CPE)**: se filtraron los documentos de la colección `datos-persona` por el campo `cobertura_medica`.

2. **Contar cuántos efectores hay en cada provincia**: mediante una operación `$lookup` entre efectores y provincias, y un posterior `$group`, se agruparon por `id_prov` y se sumaron los registros.

3. **Detectar provincias con más de dos problemas registrados**: se cruzó la colección de problemas con efectores y provincias, agrupando los resultados y filtrando por cantidad mayor a dos.

4. **Listar las capacidades de las HCE de la provincia de Mendoza**: se unieron múltiples colecciones —HCE, HCECapacidad, Capacidad, Provincia— para identificar qué capacidades están implementadas en esa jurisdicción.

5. **Identificar las provincias que tienen al menos un HCE de desarrollo propio que genera recetas digitales**: se filtraron los HCE con tipo de desarrollo `"Propio"` y generación de recetas activada, y se agruparon por provincia.

6. **Listar efectores que no tienen ningún problema registrado**: se hizo una consulta con `$lookup` y `$match` para detectar efectores que no presentan registros en la colección de problemas.

Conclusiones

En este trabajo práctico, nos encargamos del diseño de una arquitectura de datos actual, pensada para analizar los indicadores tecnológicos dentro del sistema de salud argentino. La solución que planteamos se basa en una arquitectura Lambda, que mezcla el procesamiento por lotes con el procesamiento en tiempo real, asegurando así el seguimiento histórico y la rapidez en la operación.

Para el procesamiento por lotes, usamos Apache Spark para llevar a cabo consultas, utilizando tanto el estilo MapReduce como Spark SQL. Sumamos Redis como motor de clave-valor, el cual es útil para manejar listas de tareas, y trabajar con datos temporales usando TTL y guardar de forma eficiente este tipo de registros.

También incorporamos MongoDB como base de datos orientada a documentos, lo que permitió modelar entidades con estructuras más flexibles, como efectores de salud, personas o problemas registrados. Gracias a su soporte para consultas con *pipelines de agregación*, fue posible implementar cruces complejos entre colecciones sin depender de esquemas rígidos, complementando así el enfoque relacional y facilitando un acceso más ágil a ciertos datos semiestructurados.

Además, planteamos validaciones automáticas con Great Expectations, que mejorarían la calidad de los datos procesados encontrando errores tanto en la estructura como en el significado antes de que se usen.

En resumen, la arquitectura y los flujos que proponemos no solo permiten guardar y procesar grandes cantidades de información, sino también asegurar que sea de calidad y

esté disponible cuando se necesite para los diferentes usuarios (calidad de datos, escalabilidad, consistencia y disponibilidad).

Notas adicionales

- Todos los archivos utilizados para el desarrollo de este trabajo se encuentran disponibles en un repositorio público de GitHub. En este se pueden consultar tanto los datasets originales como los archivos JSON generados para las bases no relacionales, junto con los scripts complementarios.
- Los contenedores de Docker utilizados para correr `MongoDB` y `Redis` son los mismos que empleamos durante las clases prácticas de la materia.
- Dejamos como recordatorio el DER de nuestro problema: