FACULTAD DE INGENIERÍA
UNIVERSIDAD DE LA REPÚBLICA

SISTEMA PARA LA INTEGRACIÓN DE INFORMACIÓN DE CRÍMENES

INTEGRANTES:
FACUNDO PEDRO AGÜERO SILVEIRA
MALVINA BETARTE DABOSIO

TUTORA:
MSC. ING. RAQUEL SOSA

Informe de Proyecto de Grado presentado al Tribunal Evaluador como requisito de graduación de la carrera Ingeniería en Computación

Octubre 2018

Resumen

Actualmente existe una gran variedad de canales para reportar crímenes, entre ellos se encuentran aplicaciones comunitarias, redes sociales o sistemas de denuncias del estado. Esto facilita que las personas puedan reportar crímenes de forma rápida. Como consecuencia de esta situación, aumenta de manera significativa la cantidad de información generada, por lo que es de interés lograr procesar todos estos datos heterogéneos y explotarlos en información de valor o alertas en tiempo real. Esto ayudaría en la lucha contra el crimen, la seguridad y la prevención.

El presente proyecto surge como trabajo posterior a dos artículos de investigación publicados en el año 2015. Uno de ellos propone una arquitectura que integra la información generada por usuarios, con la capacidad de análisis de GeoBI y Big Data, poniendo foco en la espacialidad de los datos. El otro artículo se basa en la arquitectura definida para proponer una solución de analíticas en tiempo real.

A partir de un análisis de la arquitectura teórica propuesta, nuestro objetivo principal es proponer una implementación concreta de la misma, por lo amplia de la propuesta original nos enfocamos en una parte, más concretamente en la ingestión, procesamiento y almacenamiento de los datos para su posterior análisis. Nos planteamos un problema de Big Data, donde se tiene en consideración la variedad de las fuentes de datos, la velocidad con la que se ingieren (necesitando una solución en tiempo real) y el volumen de los mismos.
Para resolver este problema llevamos a cabo una investigación de herramientas de Big Data con capacidad de procesamiento en tiempo real, junto a un estudio comparativo. Luego diseñamos una arquitectura tecnológica basada en productos específicos, la cual se toma como base para la implementación.
Desarrollamos un prototipo que toma datos de Twitter y aplicaciones externas a partir de web services, los cuales son procesados mediante la herramienta Apache Storm, elegida a partir del análisis comparativo realizado previamente. El procesamiento implica la normalización, validación, clasificación y cruzamientos de los datos.

Para la implementación, además de Apache Storm como herramienta de procesamiento, utilizamos Apache Kafka como cola de mensajes. Decidimos usar la tecnología Java, y se emplean como motores de base de datos PostgreSQL y MongoDB. Para validarlo se crea una web de alertas que consiste en un mapa que muestra los crímenes que ingresan al sistema con sus datos relevantes. Asimismo se diseña e implementa una solución de Business Intelligence para visualizar reportes mediante la suite de Pentaho.

Al finalizar el proyecto se logran cumplir los objetivos planteados, obteniendo luego de un estudio y comparativa de herramientas Big Data, un diseño e implementación de un sistema de integración, procesamiento y análisis de crímenes, con su validación correspondiente a través de un prototipo creado con productos específicos.

Palabras claves: Crimen, Seguridad, Big Data, Integración de datos, Procesamientos, Apache Storm.

Tabla de Contenidos

1 Introducción **4**
1.1 Motivación . . . 4
1.2 Objetivos . . . 4
1.3 Aportes . . . 4
1.4 Organización del documento . . . 5

2 Marco Teórico **6**
2.1 Conceptos básicos . . . 6
2.1.1 Crimen . . . 6
2.1.2 Seguridad Ciudadana . . . 6
2.2 Conceptos técnicos . . . 7
2.2.1 Big Data . . . 7
2.2.2 Business Intelligence (BI) . . . 9
2.2.3 GeoBI . . . 11
2.2.4 Información geográfica voluntaria . . . 11
2.2.5 Big VGI . . . 11
2.2.6 Sistema de información geográfica . . . 11
2.2.7 Bases de datos relacionales . . . 11
2.2.8 Bases de datos no relacionales . . . 12
2.2.9 Web service . . . 12
2.2.10 Application Programming Interfaces (API) . . . 12
2.2.11 Open source . . . 12
2.2.12 Machine learning . . . 12
2.3 Artículos . . . 13
2.3.1 GeoBI and Big VGI for Crime Analysis and Report . . . 13
2.3.2 Analíticas en línea de Big Data Espacial para análisis del crimen . . . 13
2.4 Trabajos relacionados . . . 13
2.4.1 CityCop . . . 13
2.4.2 Denuncias en Línea . . . 14
2.4.3 Big Data Análisis de herramientas y soluciones . . . 14
2.4.4 Observatorio Nacional sobre Violencia y Criminalidad . . . 14
2.4.5 Proyecto analíticas de video . . . 15

3 Análisis **16**
3.1 Análisis de la arquitectura teórica . . . 16
3.2 Enfoque . . . 19
3.3 Estudio de herramientas de Big Data . . . 20
3.3.1 Paradigma batch o en lotes . . . 20
3.3.2 Paradigma en tiempo real . . . 24
3.4 Comparación de herramientas de Big Data en tiempo real . . . 35
3.4.1 Ordenamiento y garantías . . . 35
3.4.2 Administración de estados . . . 36
3.4.3 Particionamiento y paralelismo . . . 36
3.4.4 Despliegue y ejecución . . . 37
3.4.5 Latencia y buffering . . . 37
3.4.6 Interoperabilidad . . . 38
3.4.7 Lenguajes de programación utilizados . . . 39
3.4.8 Comunidad . . . 39
3.5 Conclusión . . . 40

4 **Capítulo 4 - Arquitectura e Implementación** **41**
4.1 Arquitectura genérica 41
4.2 Arquitectura con productos específicos 43
4.2.1 Entrada de datos 44
4.2.2 Capa de ingestión y ETL 44
4.2.3 Salida de datos 45
4.2.4 Capa de Análisis 45
4.2.5 Capa de Visualización 45

5 **Prototipo** **46**
5.1 Implementación 46
5.1.1 Capa de Ingestión y ETL 46
5.1.2 Capas de Análisis y Visualización 59
5.2 Herramientas utilizadas 67
5.2.1 Java 67
5.2.2 Apache Maven 67
5.2.3 Eclipse 68
5.2.4 JSF 68
5.2.5 OpenLayers 68
5.2.6 GeoServer 69
5.2.7 WFS 69
5.2.8 SoapUI 69
5.2.9 Bases de datos 69
5.2.10 Kettle 70
5.2.11 GeoKettle 70
5.2.12 Mondrian 70
5.3 Pruebas del prototipo 71
5.3.1 Pruebas de integración y carga 71

6 **Gestión del proyecto** **74**
6.1 Gestión 74
6.1.1 Etapas 74
6.1.2 Desviaciones 76
6.2 Metodología 77
6.2.1 Scrum 77
6.3 Herramientas usadas 78
6.3.1 Pivotal Tracker 78
6.3.2 Gitlab 78
6.3.3 Otros 78

7 **Conclusiones y trabajo a futuro** **79**
7.1 Conclusiones 79
7.2 Trabajo a futuro 80

8 **Anexo configuración** **84**

9 **Anexo Manual de Usuario** **85**
9.1 Herramientas 85
9.2 Zookeeper 85
9.3 Apache Kafka 85
9.4 MongoDB 86
9.5 PostgreSQL 86
9.6 Crime-integration 86
9.7 Alert-integration 86
9.8 Storm 86

10 **Anexo pruebas** **87**

1 Introducción

1.1 Motivación

Actualmente existen aplicaciones comunitarias para reportar problemas de seguridad, tanto a nivel personal como de propiedades. También hay una gran cantidad de canales para denunciar crímenes e incluso sistemas de acción casi proactiva de parte del Ministerio del Interior (sistemas de cámaras, policías especializados, patrullaje, etc.). En este último tiempo surge la App 911 que además de delitos permite reportar emergencias de diferentes tipos, así como también la aplicación comunitaria CityCop la cual permite alertar delitos y recibir alertas en tiempo real de lo que ocurre en las zonas de interés del usuario.

Ante esta gran cantidad de fuentes de información es necesario poder gestionar la heterogeneidad de los datos y además de dar respuesta inmediata, poder luego analizar dicha información a más alto nivel. Este tipo de información tiene un gran componente de georeferenciación lo que habilita a realizar análisis espaciales.
Asimismo la integración de los datos y determinar el valor son temas clave. La enorme cantidad de datos disponibles debe ser ingerida y procesada para volverla útil en toma de decisiones.

Esta problemática fue trabajada en los artículos "GeoBI and Big VGI for Crime Analysis and Report"[1] y "Analíticas en línea de Big Data Espacial para análisis del crimen"[2]. En estos artículos se aborda el tema del crecimiento del uso de las plataformas VGI (Volunteered Geographic Information) y la utilidad que se le puede dar en situaciones de catástrofes, permitiendo reportar eventos a través de medios no tradicionales. En particular se enfocan en que estas plataformas están mostrando ser de gran utilidad en lo que respecta al reporte de crímenes. Un sistema que pueda recopilar esta información podría ser de gran interés para el control y la prevención en este tipo de situaciones, a través de un mejor análisis de los datos.

Estos artículos proponen una arquitectura teórica, que involucra el procesamiento de Big Data y componentes de Análisis de Business Intelligence, considerando la espacialidad de la información. Dado que esta propuesta no fue implementada, resulta de interés validar la implementación de la misma.

1.2 Objetivos

En el presente proyecto nos proponemos analizar la arquitectura propuesta, diseñar su implementación con productos concretos e implementar un prototipo de la misma.

Apuntamos a realizar el diseño tecnológico e implementación de un sistema para la integración de información de crímenes. Los objetivos específicos son:

- Realizar el estudio y análisis de la arquitectura propuesta en los artículos de referencia.
- Realizar un estudio de herramientas de Big Data.
- Diseñar una arquitectura tecnológica basada en una selección de productos específicos.
- Implementación de un prototipo de la arquitectura propuesta en base a los productos seleccionados.

1.3 Aportes

Los principales aportes de este proyecto son:

- Conocimientos adquiridos a través del estudio de herramientas de Big Data y su comparación.
- El diseño de la arquitectura tecnológica para la implementación del sistema.
- Lograr un estudio de factibilidad y validación de la implementación de la arquitectura teórica propuesta en los artículos de referencia mediante la implementación de un prototipo.

1.4 Organización del documento

Este documento está divido en capítulos los cuales se detallan a continuación.

En el capítulo 1 se da una introducción, donde se plantea la motivación, los objetivos principales y aportes logrados en el desarrollo del proyecto.

En el capítulo 2 se definen los conceptos necesarios para la comprensión del informe y del trabajo realizado. Se describen los artículos de referencia que se toman como base para la investigación, y se presentan trabajos relacionados.

En el capítulo 3 se detalla el estudio de la arquitectura teórica la cual sirve para comprender el problema, y se explica la decisión tomada en cuanto al enfoque de la solución.
Asimismo se realiza un estudio de herramientas de Big Data, junto a un estudio comparativo.

En el capítulo 4 se presenta una arquitectura genérica enfocada en nuestro problema a resolver, y luego se muestra la arquitectura basada en productos específicos, junto a una explicación de la función de cada etapa.

El capítulo 5 describe la implementación del prototipo logrado. Incluye las decisiones tomadas en la implementación y el funcionamiento de los componentes principales. Además, se incluyen las herramientas utilizadas, y las pruebas de sistema realizadas.

El capítulo 6 muestra los puntos principales en la gestión del proyecto, incluyendo las estimaciones iniciales y finales, la descripción de las etapas, las desviaciones del proyecto y las herramientas de apoyo.

El capítulo 7 contiene las conclusiones y trabajo a futuro.

2 Marco Teórico

Este capítulo tiene como objetivo explicar los conocimientos previos relevantes, necesarios para la comprensión y realización del proyecto, como por ejemplo el término Big Data. Asimismo se describen brevemente los dos artículos que se toman como referencia y punto de partida para el desarrollo del trabajo, y se muestran algunos ejemplos de trabajos relacionados.

2.1 Conceptos básicos

2.1.1 Crimen

La Real Academia Española [3] define a un crimen como un delito grave, una acción indebida o reprensible, o una acción voluntaria de matar o herir gravemente a alguien.
Uno de los puntos más importantes para detener y prevenir los crímenes son las denuncias, pero no todas las personas las llevan a cabo por el tiempo que insumen. Sin embargo, gracias al desarrollo de la tecnología y el fácil acceso a Internet, actualmente los delitos pueden ser reportados fácilmente y logran tener un accionar más rápido.
Hoy en día, la sociedad actúa de diferentes maneras para prevenir, detener y alertar los crímenes que se puedan cometer en su ciudad, debido a la gran variedad de canales existentes de fácil y rápido acceso.

El Estado Uruguayo, a través del Ministerio del Interior, ha asumido un rol importante ofreciendo diferentes herramientas para que las personas puedan alertar un crimen de cualquier índole en el momento en que ocurre. Dentro de estas herramientas se encuentran los sistemas en línea como por ejemplo el sistema de Denuncias en Línea o la App de 911, donde se requiere de la intervención del individuo que es partícipe o espectador del crimen para generar la alerta.
Se han implementado e instalado cámaras de seguridad mayoritariamente en la ciudad de Montevideo, por parte del Ministerio del Interior y la Intendencia de Montevideo, que trabajan conjuntamente (mediante un acuerdo) compartiendo las imágenes e incorporando a través del Ministerio un sistema analítico de imágenes por comportamiento que permite identificar y alertar a la Policía sobre actitudes de personas, cambios de velocidades o sentido de circulación de vehículos, así como detección de peatones y niveles de sonido. [4]

A su vez, han surgido aplicaciones colaborativas donde las personas reportan hechos delictivos y reciben alertas de zonas de interés, como por ejemplo la aplicación CityCop[5]. Esto permite al ciudadano tener información de crímenes o actividades sospechosas en el momento, permitiendo tomar los recaudos necesarios.

2.1.2 Seguridad Ciudadana

La seguridad ciudadana es la acción integrada que desarrolla el Estado, con la colaboración de la ciudadanía y de otras organizaciones de bien público y/o privado. Está destinada a asegurar la convivencia y desarrollo pacífico, la erradicación de la violencia, la utilización pacífica y ordenada de vías y de espacios públicos y, en general, evitar la comisión de delitos y faltas contra las personas y sus bienes.

La seguridad ciudadana es una modalidad específica de la seguridad humana, que puede ser definida inicialmente como la protección universal contra el delito violento o predatorio. En este sentido, seguridad ciudadana es la protección de ciertas opciones u oportunidades de todas las personas (su vida, su integridad, su patrimonio) contra un tipo específico de riesgo (el delito) que altera de forma “súbita y dolorosa” la vida cotidiana de las víctimas. [6]

2.2 Conceptos técnicos

2.2.1 Big Data

En las últimas décadas, la cantidad de información creada ha aumentado significativamente. Más de 30.000 gigabytes de datos son generadas cada un segundo, y la creación de los mismos se va acelerando con el paso del tiempo. A su vez, los datos que se manejan son diversos, provenientes de blogs, redes sociales, fotos, entre otros. [7]

Este crecimiento exponencial de la información ha afectado profundamente a las empresas, ya que los sistemas de bases de datos tradicionales, como las bases de datos relacionales, han llegado a su límite, haciendo que se quiebren bajo las presiones de "Big Data". Los sistemas tradicionales y las técnicas de administración de datos asociadas a ellos, no han logrado escalar a Big Data.

Ante este escenario, una nueva generación de tecnologías han aparecido, muchas de ellas consideradas bajo el termino de "noSQL". Estos nuevos sistemas pueden escalar a conjuntos de datos mucho más grandes, pero a costa de un conjunto de técnicas nuevas.
Muchos de estos sistemas de Big Data fueron iniciados por Google, entre los que se incluyen los sistemas de archivos distribuidos, MapReduce, Hadoop, entre otros. Esto dio inicio a una gran cantidad de proyectos.

Big Data es un término amplio y con definiciones competitivas.
Michael Stonebraker, científico especializado en bases de datos, considera que Big Data quiere decir al menos tres cosas: gran volumen, gran velocidad y gran variedad. [8]

- Volumen: un sistema Big Data es capaz de almacenar una gran cantidad de datos mediante infraestructuras escalables y distribuidas. En los sistemas de almacenamiento de hoy en día, comienzan a aparecer problemas de rendimiento al tener cantidades de datos del orden de magnitud de petabytes o superiores. Big Data está pensado para trabajar con estos volúmenes de datos.
  Hace referencia a análisis SQL simples pero con petabytes de datos y cientos de nodos, o análisis no-SQL complejos como por ejemplo operaciones matemáticas complejas (machine learning, clustering, trend detection, entre otros) en cantidades muy grandes de datos.

- Velocidad: una de las características más relevantes es el tiempo de procesado y respuesta sobre estos grandes volúmenes de datos, obteniendo resultados en tiempo real y procesándolos en tiempos muy reducidos. No sólo se trata de procesar sino también de la recepción, ya que hoy en día las fuentes de datos pueden llegar a generar mucha información cada segundo, obligando al sistema receptor a tener la capacidad para almacenar dicha información de manera muy veloz.
  Por lo tanto, gran velocidad refiere a la capacidad de absorber y procesar una gran cantidad de datos entrantes como ser redes sociales, sensores, aplicaciones, entre otros.

- Variedad: las nuevas fuentes de datos proporcionan nuevos y distintos tipos y formatos de información a los ya conocidos hasta ahora, como datos no estructurados, que un sistema Big Data es capaz de almacenar y procesar sin tener que realizar un preproceso para estructurar o indexar la información.

Más adelante se agregan dos Vs a las características de Big Data: Veracidad y Valor.

- Veracidad: se refiere a la incertidumbre de los datos, es decir, al grado de fiabilidad de la información recibida. Es necesario invertir tiempo para conseguir datos de calidad, aplicando soluciones y métodos que puedan eliminar datos imprevisibles que puedan surgir. Es indispensable entender si los datos que se están procesando tienen un significado válido para el problema que se analiza.

- Valor: el objetivo final es generar valor de toda la información almacenada a través de distintos procesos de manera eficiente y con el coste más bajo posible.

Según Guy Harrison, en su libro "Next Generation Databases" [9], hay dos cambios significativos complementarios en el papel de los datos dentro de la informática y la sociedad: hay más datos, es

decir que ahora tenemos la habilidad de almacenar y procesar los mismos ya sea multimedia, de redes sociales o datos transaccionales, en su formato original y mantener estos datos potencialmente a perpetuidad. A su vez, esto también permite generar mayor valor a partir de la información, gracias a los avances de Machine Learning, análisis predictivos e inteligencia predictiva.

En Big Data los datos se pueden dividir en tres tipos:

- Datos estructurados: es información ya procesada, filtrada y con un formato estructurado. Es el tipo de datos que más se usan actualmente, sobre todo en bases de datos relacionales.
- Datos semi-estructurados: es información procesada y con un formato definido pero no estructurado. De esta manera se puede tener la información definida con una estructura variable. Ejemplos de esto son las bases de datos basadas en columnas y los archivos del tipo HTML o XML.
- Datos no estructurados: corresponde a información sin procesar y que puede tener cualquier estructura. Se puede encontrar en cualquier formato: texto, vídeo, imagen, código, entre otros. Los directorios de logs de aplicaciones o la información proveniente de redes sociales son buenos ejemplos de datos no estructurados.

Una de las características principales de un sistema Big Data es el de trabajar con datos no estructurados.

En la figura 1 se pueden observar algunas herramientas de Big Data existentes en la actualidad y como se pueden clasificar según su función.
La categoría principal Big Data son los motores de procesamiento, encargados de tomar los datos que entran y procesarlos de manera distribuida, realizando funciones de filtrado, agregación o preparación para su análisis. Otra categoría es el almacenamiento, cuya función es guardar y gestionar grandes volúmenes de datos donde éstos pueden ser no estructurados. Para la gestión de recursos existen tecnologías diseñadas para las tareas de planificación y asignación de los recursos dentro del cluster, en donde se realizará el procesamiento. A su vez, se cuenta con tecnologías de mensajería, como colas de mensajes, que permiten el intercambio eficiente de datos con otros sistemas.

Para simplificar el acceso a los datos y realizar las consultas de forma eficiente, existen bibliotecas de consulta, así como también bibliotecas de Machine Learning, que implementan algoritmos para la clasificación, regresión, filtrado, predicción, entre otros.[10]

Fig. 1: Clasificación herramientas de Big Data

2.2.2 Business Intelligence (BI)

Business Intelligence (BI) refiere a conceptos y métodos para mejorar la toma de decisiones de negocio utilizando sistemas de soporte basados en hechos.

Es el conjunto de metodologías, aplicaciones y tecnologías que permiten reunir, depurar y transformar datos de los sistemas transaccionales e información desestructurada (interna y externa a la compañía) en información estructurada, para su explotación directa (reporting, análisis OLTP / OLAP, alertas...) o para su análisis y conversión en conocimiento, dando así soporte a la toma de decisiones sobre el negocio. [11]

2.2.2.1 Datawarehouse

Un datawarehouse es una base de datos corporativa que se caracteriza por integrar y refinar información de una o varias fuentes. Luego se procesa la información permitiendo de esta manera su análisis desde infinidad de perspectivas y con grandes velocidades de respuesta. [12]

2.2.2.2 *Sistema de Datawarehousing*

Es un sistema informático capaz de ofrecer información para la toma de decisiones, y cuyo componente principal es un Data Warehouse.

La arquitectura típica de un sistema de este tipo se presenta en la figura 2, en la cual se distinguen las siguientes capas: la capa de fuentes de datos donde se consideran las posibles entradas, la capa de backend encargada de procesamiento ETL. La capa de datawarehouse es la que contiene la base de datos, la capa de OLAP con los datos multidimensionales provenientes del datawarehouse y el servidor OLAP, y por último, la capa de frontend con las herramientas clientes que permiten a los usuarios explotar los contenidos del datawarehouse.

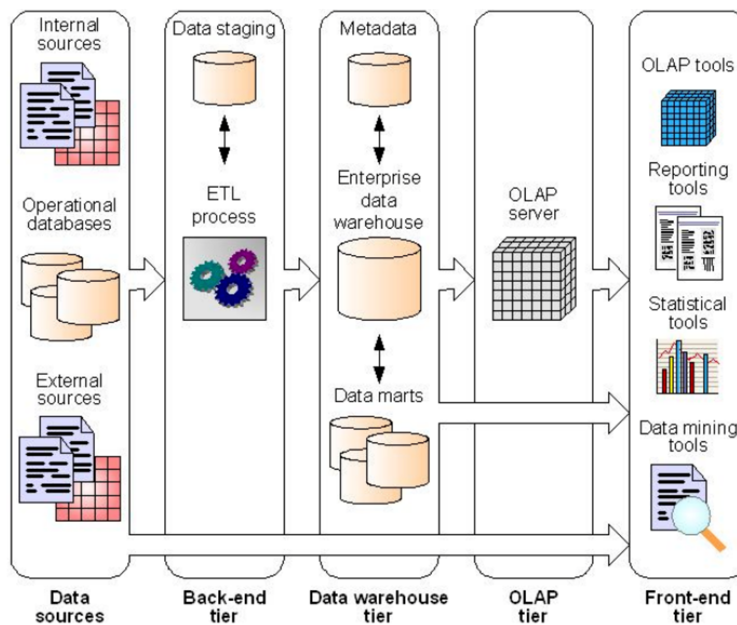


Fig. 2: Arquitectura de un Sistema de Datawarehousing

El desarrollo de un sistema de esta índole está compuesto por cuatro fases principales: especificación de requerimientos, diseño conceptual, diseño lógico, y diseño físico e implementación. Entre los componentes elementales a desarrollar se encuentran la adquisición de datos, el almacenamiento del datawarehouse y los mecanismos de acceso por parte de usuarios. [13]

2.2.2.3 *ETL*

Extract, Transform, Load (Extraer, transformar, cargar) es el proceso que permite mover los datos de múltiples fuentes, darles un formato, limpiarlos, y por ultimo cargarlos en otra base de datos o datawarehouse para su posterior análisis.

- Extracción: adquisición de la información de distintas fuentes tanto internas como externas.
- Transformación: filtrado, limpieza, depuración, homogeneización y agrupación de la información.
- Carga: organización y actualización de los datos y los metadatos en la base de datos.

2.2.2.4 *OLAP*

Los sistemas OLAP son bases de datos orientadas al procesamiento analítico. Suele implicar la lectura de grandes cantidades de datos para lograr extraer algún tipo de información útil, por ejemplo tendencias de ventas, patrones de comportamiento de consumidores, elaboración de informes complejos, entre otros. El objetivo principal es agilizar las consultas de grandes volúmenes

de datos.
Las bases de datos OLAP se suelen alimentar de información proveniente de los sistemas operacionales existentes, a través del proceso de extracción, transformación y carga (ETL).

2.2.2.5 Dashboard

Un dashboard es una herramienta de visualización de datos que muestra el estado actual de las métricas y los indicadores clave de rendimiento (KPI) para una organización. Los paneles consolidan y organizan números, métricas y, en ocasiones, tablas de puntuación de rendimiento en una sola pantalla. Se pueden adaptar para un rol específico y mostrar métricas dirigidas a un único punto de vista o departamento. Las características esenciales de un dashboard incluyen una interfaz personalizable y la capacidad de extraer datos en tiempo real de múltiples fuentes.

2.2.3 GeoBI

GeoBI es la extensión del área Business Intelligence en la que se agrega la interacción con mapas y elementos geoespaciales.
Surge en la búsqueda de enriquecer la toma de decisiones considerando la dimensión espacial de la información. Consecuentemente el concepto de OLAP pasa a ser spatial OLAP (SOLAP) y datawarehouse a Spatial datawarehouse (SDW). [14]

2.2.4 Información geográfica voluntaria

La información geográfica voluntaria (VGI) es un tipo especial de contenido generado por el usuario. Hace referencia a la información geográfica recopilada y compartida voluntariamente por el público en general.
Michael F. Goodchild fue el que definió este término en 2007 en un artículo y para él, la VGI combina elementos de la Web 2.0, la inteligencia colectiva y la neogeografía. Se puede decir que es la información geográfica creada o recogida por voluntarios de forma organizada, no necesariamente especialistas y publicada online, comúnmente con el fin de usarla en proyectos de bien común. [15]

2.2.5 Big VGI

En los últimos años, la aparición de medios masivos y aplicaciones a las cuales puede acceder el público en general, hizo que la información geográfica voluntaria crezca de manera exponencial. Gracias a este fenómeno se presenta la oportunidad de muchos usos potenciales, pero a su vez existen algunos desafíos y problemas derivados de la generación de volúmenes nunca pensados de información geográfica, a este problema se le denomina Big VGI. Por lo general, las soluciones de Big Data consisten en mejorar la interoperabilidad, descubrimiento y análisis de grandes volúmenes de datos, sin embargo los datos producidos por VGI tiene diferencias con los datos tradicionales por lo que deben tenerse en cuenta tales diferencias a la hora de procesarlos.

2.2.6 Sistema de información geográfica

Un sistema de información geográfica o SIG por sus siglas, es un conjunto de herramientas que integran y relacionan diversos componentes que permiten la organización, manipulación, análisis y modelado de grandes cantidades de datos de la realidad que están vinculados a una referencia espacial. Los SIG son herramientas que permiten a los usuarios crear consultas interactivas, analizar la información espacial, editar datos, mapas y presentar los resultados de todas estas operaciones.

2.2.7 Bases de datos relacionales

Una base de datos relacional es una base de datos que se trata como un conjunto de tablas y se manipula de acuerdo a un modelo de datos relacional. La interfaz estándar para trabajar con este tipo de bases de datos es el lenguaje de programación SQL (Structured Query Languaje). SQL es un lenguaje estándar que permite realizar consultas con el objetivo de recuperar información de las bases de datos de manera sencilla. [16]

2.2.8 Bases de datos no relacionales

Las bases de datos no relacionales o noSQL son bases de datos que no utilizan el modelo relacional para modelar sus datos. Se originaron para enfrentar el desafío del tratamiento de datos que las bases de datos estructuradas no solucionaban. Responden a la necesidad de escalabilidad horizontal que tiene cada vez más las organizaciones y pueden manejar enormes volúmenes de datos.

2.2.9 Web service

Un web service permite a distintas aplicaciones, en diferentes orígenes, comunicarse entre ellas sin necesidad de escribir programas muy complejos. Es una colección de protocolos abiertos y estándares usados para intercambiar datos entre aplicaciones o sistemas. Los web services no están ligados a ningún sistema operativo ni a lenguajes de programación, por ejemplo un programa escrito en Java puede comunicarse con otro escrito en Perl, así como también aplicaciones Windows pueden comunicarse con aplicaciones Unix.

Dentro de las ventajas de usar web service se tiene que aportan interoperabilidad entre las aplicaciones de software, independientes de sus propiedades o de las plataformas sobre las que se instalan, fomentan los estándares y protocolos basados en texto que hacen más fácil acceder a su contenido y entender su funcionamiento, y permiten que servicios o software de diferentes compañías ubicadas en diferentes lugares puedan ser combinados fácilmente para proveer servicios integrados. [17]

2.2.9.1 SOAP

Simple Object Access Protocol (SOAP) es un protocolo estándar que define como los objetos de diferentes procesos puedan comunicarse por medio de intercambio de mensajes XML.

2.2.10 Application Programming Interfaces (API)

Se demonina API (interfaces de programación de aplicaciones) a un conjunto de comandos, funciones y protocolos que se utilizan para que un módulo de software específico pueda comunicarse con otro. Por ejemplo la API Twitter4j tiene determinadas operaciones para que aplicaciones desarrolladas en Java puedan integrarse con Twitter de una manera más fácil.

2.2.11 Open source

El software open source o de código abierto refiere a cualquier programa cuyo código fuente se pone a disposición para su uso o modificación, conforme los usuarios u otros desarrolladores lo consideren conveniente. El software de código abierto, por lo general, se desarrolla como una colaboración pública y se hace disponible de manera gratuita.

2.2.12 Machine learning

Machine learning es una disciplina científica del ámbito de la inteligencia artificial con el propósito de crear sistemas que aprenden automáticamente. En este contexto, aprender quiere decir identificar patrones complejos en millones de datos. La máquina aprende un algoritmo que analiza los datos y es capaz de predecir comportamientos futuros. Automáticamente implica que estos sistemas se mejoran de forma autónoma con el tiempo, sin intervención humana. [18]

2.3 Artículos

Este proyecto surge como trabajo posterior a los artículos:
"GeoBI and Big VGI for Crime Analysis and Report"[1], el cual propone una arquitectura para manejar el análisis y reporte de crímenes a partir de información geográfica voluntaria, y
"Análiticas en línea de Big Data Espacial para análisis del crimen" [2] que propone una primera solución para el procesamiento en tiempo real de los datos proveniente de información geográfica voluntaria.

2.3.1 GeoBI and Big VGI for Crime Analysis and Report

En la era de la Web 2.0 en la que estamos, en donde los ciudadanos pueden denunciar los delitos directamente desde la Web, se tiende a denunciar hasta los más pequeños. Además, hay algunas plataformas de VGI especializadas en denuncias e información sobre delitos. El uso de toda esta información integrada podría ayudar a mejorar la lucha contra el crimen, la prevención, e incluso las estrategias que los ciudadanos utilizan para estar más seguros.

En este trabajo se propone el uso de VGI y open Data en el dominio de reportes de crímenes. Se definen técnicas de GeoBI para mostrar los datos de diferentes tipos de usuarios con diferentes propósitos, como por ejemplo agencias gubernamentales que trabajan en la resolución y prevención de delitos, ciudadanos que pueden reportar crímenes y además tomar recaudos a partir de la información que puedan obtener.
Además, se propone una arquitectura que integra el acceso a datos geoespaciales y VGI con la capacidad de análisis de GeoBI y Big Data basada en el MAD Framework. Más adelante se profundiza en esta arquitectura.

2.3.2 Analíticas en línea de Big Data Espacial para análisis del crimen

Este trabajo está enfocado en la utilización de herramientas de Big Data Espacial basadas en algoritmos en línea para la detención de incidencias criminales en tiempo real, especialmente a partir de información de sensores como pueden ser cámaras de videovigilancia y la web social, por ejemplo Twitter.

En este artículo se toma la arquitectura propuesta en el artículo anterior y se propone profundizar en la misma. Específicamente, las herramientas propuestas involucran las capas de ingestión de datos y ETL así como la capa de análisis.
Se propone una solución que contempla las distintas "V"s del modelo de Big Data (variedad, velocidad, volumen, veracidad y valor). A partir de un análisis de herramientas de Big Data se opta por la utilización de Apache Storm para resolver el problema planteado ya que aporta a la solución la capacidad de lidiar con el aumento de la complejidad y volumen de los datos analizados.

2.4 Trabajos relacionados

2.4.1 CityCop

CityCop es una plataforma de alertas comunitaria, que tiene como objetivo combatir la delincuencia. Permite alertar delitos y recibir alertas en tiempo real de lo que ocurre en las zonas de interés del usuario. La aplicación funciona de la siguiente manera: se debe descargar la app en un smartphone y registrarse como usuario indicando los datos personales. Luego, se marcan las zonas de interés (casa, trabajo, entre otros), de las cuales se va a recibir información de lo que sucede en tiempo real. Para realizar una denuncia hay que georreferenciar el evento, esto hace que CityCop sea una app de VGI.

Algunos ejemplos de tipos de denuncias que se pueden hacer son: robo a casa, persona, vehículo o comercio, actividad sospechosa, homicidio, acto vandálico, denuncia de boca de venta de droga, entre otros.

La aplicación no está pensada para que a partir de una denuncia haya una respuesta policial, sino que el objetivo es informar a los ciudadanos (usuarios) de diferentes eventos que se dan a diario en materia de seguridad, ya que no todas las víctimas de delitos realizan la denuncia de manera

formal. Tampoco aparecen en los medios todos los delitos, por lo que muchas veces es dificil saber lo que pueda ocurrir a nuestro alrededor. [5]

En una nota al diario El Observador uno de los creadores de CityCop, Federico Cella, dijo: "Buscamos hacer una aplicación que replicara lo que pasa en la vida real; va más allá de lo que son los medios y las autoridades, tiene más que ver con la interacción de los ciudadanos" cuando hablaba sobre el objetivo de la app. La aplicación tiene más de 100.000 descargas en todo el mundo. [19]

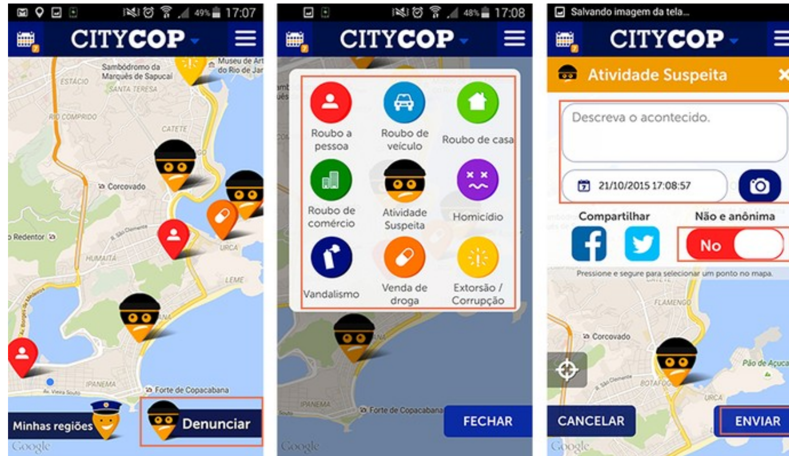


Fig. 3: Interfaz gráfica de CityCop

2.4.2 Denuncias en Línea

De un tiempo a esta parte ya es posible realizar denuncias en línea. No es necesario tener que trasladarse hasta una Seccional Policial para poder realizarla. El sistema de denuncias en línea es sencillo y solo pide algunos datos personales, dirección, y la descripción del hecho denunciado. El sistema hace una validación de la dirección antes de dar de alta la denuncia, esto hace que se tenga un dato real y válido. Etnre las ventajas de contar con este servicio se puede ver que las personas anteriormente ante un hecho no tan grave no se molestaban en realizar la denuncia, por el simple hecho de que requería tiempo y moverse hasta una Seccional Policial para poder realizarlo. [20]

2.4.3 Big Data Análisis de herramientas y soluciones

"Big Data Análisis de herramientas y soluciones"[21] es un proyecto de fin de carrera de los estudios de Ingeniería Superior en Informática, realizado por Roberto Serrat Morros y Jordi Sabater Picañol de la Facultat d'Informática de Barcelona - UPC en Noviembre de 2013. Este proyecto realiza un estudio de la situación en la que se encontraba el campo Big Data en ese momento, a su vez se lleva a cabo un estudio técnico de soluciones de Big data y una implementación de un caso de uso mediante herramientas de la misma.

Los objetivos principales del proyecto son realizar un estudio teórico sobre Big Data, y diseñar e implementar dos ejercicios pilotos con el propósito de comparar dos soluciones.

2.4.4 Observatorio Nacional sobre Violencia y Criminalidad

El Observatorio Nacional sobre Violencia y Criminalidad [22], es un proyecto del Ministerio del Interior lanzado en Agosto de 2005. Fue creado con el objetivo de brindar datos confiables y reales, que se elaboran mediante una metodología rigurosa para el tratamiento de los principales indicadores sobre la evolución de la criminalidad y la gestión policial en Uruguay.

Además de su función estadística, donde la información se actualiza trimestralmente, el Observatorio se asume como un proyecto en movimiento para promover la modernización de los sistemas

de información, la revisión y ampliación de las problemáticas a diagnosticar y la producción de conocimiento y análisis sobre los asuntos de la violencia, la criminalidad y la inseguridad en el Uruguay.

Este Observatorio persigue los siguientes objetivos:

- Centralizar, procesar y analizar la información estadística sobre violencia y criminalidad que produce el Ministerio del Interior.
- Aplicar criterios espaciales y temporales estandarizados para la medición confiable de distintos fenómenos.
- Actualizar trimestralmente los datos para mantener informada a la población acerca de la evolución de los principales indicadores en materia de violencia y criminalidad.
- Profundizar el trabajo sectorial dentro del Sistema Estadístico Nacional para el intercambio de información que estimule la producción de conocimiento original en las ciencias sociales.
- Estimular, a través de la difusión pública, la integración de los distintos esfuerzos en materia de estudios y análisis sobre la evolución y los perfiles de la violencia y la criminalidad en el Uruguay.

2.4.5 Proyecto analíticas de video

El proyecto de grado de nombre "Algoritmos de inteligencia computacional para la detección de patrones de movimiento de personas" [23], fue desarrollado en el marco de la materia Proyecto de Grado de Facultad de Ingeniería de la Universidad de la República, en el año 2016.
A partir del crecimiento del número de cámaras de vigilancias instaladas en la ciudad, los centros de gestión y control de sistemas de vigilancia han aumentado de tamaño y costo. Actualmente el procesamiento de las imágenes de dichas cámaras son realizadas mayormente por personas, por lo que el área de detección de hechos anómalos a partir de procesos automáticos es un área aún por explotar.

Por lo tanto se presenta un estudio de procesamiento de imágenes y detección de patrones, así como también una prueba de concepto de un sistema capaz de generar alertas ante hechos predefinidos que ocurren en una secuencia de imágenes, por ejemplo un robo.
La salida de estas alertas es un archivo de log, por lo que vimos la potencial integración de dicho sistema con nuestra plataforma, como forma de integrarnos a sensores.

3 Análisis

Comenzamos analizando la arquitectura propuesta en el artículo "GeoBI and Big VGI for Crime Analysis and Report" [1], con el fin de desarrollar una solución basada en productos específicos a partir de la misma.
Luego de este análisis, presentamos el enfoque de nuestro proyecto, y a partir del mismo realizamos un estudio y comparativa de herramientas para poder diseñar una arquitectura específica.

3.1 Análisis de la arquitectura teórica

Este proyecto se apoya en el artículo "GeoBI and Big VGI for Crime Analysis and Report" [1], que tiene como objetivo mostrar como el área de Business Intelligence brinda herramientas para interpretar la gran cantidad de información que se genera a través de VGI, tomando el caso de uso de reporte de crímenes. Asimismo, se propone una arquitectura que integra el acceso a datos geoespaciales y VGI, con la capacidad de análisis de GeoBI y Big Data basada en un Framework en particular.

Actualmente la información geográfica voluntaria (VGI) ha incrementado la diversidad de información georreferenciada que está disponible en línea. Ofrece una nueva noción de infraestructura para recolectar, sintetizar, verificar y redistribuir datos geográficos a través de tecnologías de geolocalización, dispositivos móviles y bases de datos geográficas.
En los últimos años, se experimenta un crecimiento explosivo en la disponibilidad de VGI, por lo que requiere de modelos de almacenamiento escalables para manejar conjuntos de datos espaciales a gran escala, mejorando la calidad del servicio de datos con respecto a la precisión, la credibilidad, la confiabilidad y el valor general.

En la era de la Web 2.0 en la que estamos, en donde los ciudadanos pueden denunciar los delitos directamente desde la Web, se tiende a denunciar hasta los más pequeños. Además hay algunas plataformas de VGI especializadas en denuncias e información sobre delitos. El uso de toda esta información integrada podría ayudar a mejorar la lucha contra el crimen, la prevención, e incluso las estrategias que los ciudadanos utilizan para estar más seguros.

Este artículo propone el uso de VGI y open Data en el dominio de reportes de crímenes, además de técnicas de GeoBI para mostrar los datos de diferentes tipos de usuarios con diferentes propósitos. Entre dichos usuarios se identifican agencias gubernamentales que trabajan en la resolución y prevención de delitos, ciudadanos que pueden reportar crímenes, y además tomar recaudos a partir de la información que puedan obtener.
En la propuesta de la plataforma se identifican cuáles serían estos usuarios, considerados como consumidores y productores. Los consumidores se identifican como ciudadanos, policías, bomberos, unidades de emergencia móviles, analistas y gerentes que estén interesados en la plataforma. Los productores serían usuarios expertos y gerentes cuyos intereses tienen que ver con estudiar cómo localizar centros de emergencia, estaciones de bomberos, patrullas, etc. y también manejar la logística de los múltiples insumos.

Los requerimientos se resuelven considerando el MAD Framework (Monitor, Analyze and Drill to detail) el cual separa las necesidades de los usuarios en capas según su perfil. Es un framework de análisis top-down que permite diseñar las interacciones poniendo el enfoque en los aspectos de BI. Se divide en tres capas:

- Monitor: ofrece una visión general resumida y gráfica de los datos pensada para usuarios empresariales, por ejemplo portales web, dashboards o reportes.
- Analyze: permite a los usuarios explorar los indicadores (KPIs) desde múltiples perspectivas o dimensiones utilizando filtros, ejemplo de esto son los cubos SOLAP, Data Mining, módulos de Machine Learning. Se puede profundizar en estos indicadores y está pensado para managers y analistas.
- Drill: ofrece datos detallados para los usuarios donde pueden explorar datos de nivel atómico en el datawarehouse o en el sistema fuente para obtener información importante, por ejemplo consultas dinámicas e interactivas directamente al datawarehouse, análisis de sentimientos.

En la figura 4 se presenta un gráfico de este framework.

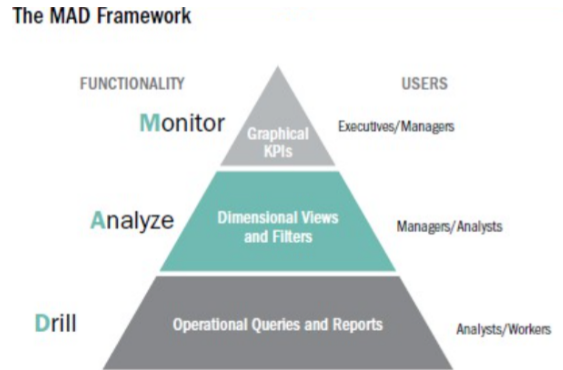


Fig. 4: MAD Framework

En el desarrollo del trabajo se plantea una tabla en la cual se listan los potenciales usuarios en cada capa y las herramientas propuestas. Por ejemplo en la capa de Monitoreo se identifican como potenciales usuarios a los directores de planificación de diferentes agencias, por ejemplo: personal del Instituto Nacional de Estadística (INE), gerentes de ministerios, etc. Éstos contarían con acceso al portal web, dashboards y reportes con gráficos, entre otros.
En la capa de Análisis se consideran a los analistas del Ministerio del Interior (policías, bomberos) que usarían cubos OLAP, Machine Learning, entre otros.
En la capa de Drill se identifican al personal responsable de asignar recursos a emergencias (policía, bomberos, etc.), para los que se necesitan consultas dinámicas e interactivas al datawarehouse, módulos de alertas que generen eventos que requieran de atención inmediata.

La arquitectura propuesta para el análisis y reporte de crímenes se presenta en la figura 5

La solución plantea comenzar construyendo una plataforma mínima de VGI y luego ir añadiendo capacidades de BI.
Inicialmente esta plataforma debe contar con capacidades GIS para soportar entradas de datos de usuarios, por ejemplo desde una aplicación web o móvil, y debe contar con servidores de mapas, bases de datos geográficas, entre otros.

A continuación se describen las distintas capas de la arquitectura, la cual integra información geográfica voluntaria y el acceso de datos geoespaciales, con las capacidades de análisis de GeoBI y Big Data:

- Capa de ingestión y ETL: Es la capa responsable del acceso, integración y almacenamiento de los datos, así como también de las operaciones tradicionales de BI (Extracción, Transformación y Carga) para la carga del datawarehouse. Soporta diferentes entradas de datos dentro del contexto de crímenes, como ser datos de aplicaciones web, móviles, bases de datos institucionales y geoservicios web espaciales.
  Esta capa debe validar, limpiar, transformar y reducir la información para posterior procesamiento.

- Capa de Análisis: Incluye el análisis espacial OLAP y las herramientas matemáticas específicas del ambiente de Big Data.
  Debe tener en cuenta el conocimiento aprendido en las fases de respuesta y recuperación del ciclo de gestión de emergencias. También debe incluir planes de respuestas predefinidas creadas en la etapa de Planificación. La combinación de los datos de las respuestas de emergencia debe estar preparada para desencadenar una respuesta inmediata.

- Capa de Visualización: Provee distintas interfaces de usuario con reportes, toma de decisiones en tiempo real y dashboards geográficos, así como también sistemas de alertas.

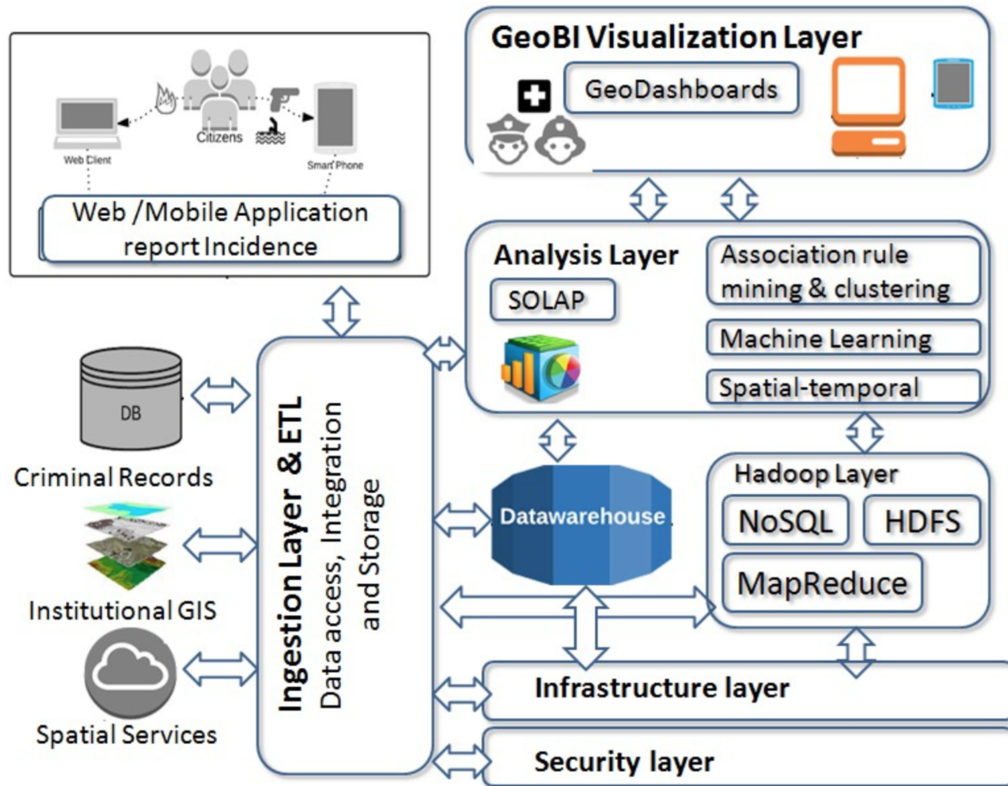


Fig. 5: Arquitectura

- Capa de Infraestructura: Es la capa física que tiene el almacenamiento. Es importante para la escalabilidad y funcionamiento de la arquitectura de Big Data.
- Capa de Seguridad: Es de suma importancia en el caso de crímenes contar con una capa de seguridad, con métodos de autorización y autenticación sobre los datos.

La arquitectura propuesta está basada en principios de interoperabilidad, donde las interfaces entre capas son estándares.

3.2 Enfoque

A partir del estudio de la arquitectura presentado anteriormente, observamos que la misma es compleja y abarca muchas capas. Presenta nuevas áreas de investigación como por ejemplo Machine Learning, GeoBI, Hadoop, bases de datos no relacionales, entre otros.

Por lo tanto decidimos enfocarnos en una parte de la arquitectura, concretamente en la capa de ingestión y ETL.
Los datos que ingresan a esta capa provienen de diversas fuentes heterogéneas, por lo que se debe considerar la **variedad** de las mismas. Es decir, es necesario poder integrarse con sistemas de información de VGI, sensores, redes sociales, archivos de texto, entre otros.
Nos interesa considerar la **velocidad** en la que se ingieren y procesan los datos, por lo tanto se busca una solución en tiempo real.
A raíz de que las fuentes de datos mencionadas anteriormente generan en la actualidad una gran cantidad de datos, se debe tener capacidades para procesar este **volumen**.

Al considerar la variedad, velocidad, y volumen, nos encontramos ante un problema de Big Data, siendo éstas las tres "V" principales. Actualmente Big Data considera la veracidad y el valor como parte importante de su definición, en efecto son tomadas en consideración en el dominio del problema. La **veracidad** de los datos es tenida en cuenta a través del cruzamiento de los mismos y la validación a partir de cierto motor de reglas. Por último, a partir de la información procesada se genera un sistema de datawarehousing utilizado posteriormente para generar información de **valor**.

Para enfrentar este problema, comenzamos con un estudio de las distintas herramientas de Big Data, basándonos fuertemente en la solución propuesta en el artículo "Analíticas en línea de Big Data Espacial para análisis del crimen". En primera instancia fue necesario investigar las herramientas de procesamiento existentes, para luego elegir algunas de ellas para su estudio en profundidad.
Aprendimos los aspectos generales de cada una de ellas y llevamos a cabo un estudio comparativo teniendo en cuenta los siguientes aspectos: ordenamiento y garantías, administración de estados, particionamiento y paralelismo, despliegue y ejecución, latencia y buffering, interoperabilidad, lenguajes soportados y comunidad.

A partir del resultado de esta investigación, diseñamos una arquitectura tecnológica genérica focalizada en la parte que nos es de interés. Es decir, en la etapa de ingestión y procesamiento de los datos. Esta arquitectura debe contemplar las distintas fuentes de datos, la ingesta de mensajes en tiempo real, el procesamiento de los mismos, el almacenamiento y posterior análisis y visualización.
Posteriormente seleccionamos los distintos componentes y herramientas que constituyen la arquitectura, para realizar una implementación de un prototipo.
Luego definimos las entradas de datos a considerar en el prototipo, diseñamos el flujo de procesamiento y determinamos las salidas de datos. A su vez, al tener que integrarnos con otros sistemas, exploramos las diversas posibilidades y mejores prácticas.
Finalmente implementamos el prototipo exitosamente y realizamos pruebas para evaluar el comportamiento del mismo con distintas configuraciones, buscando optimizar el procesamiento.

En resumen, el presente proyecto se propone el diseño de un sistema de integración, procesamiento y análisis de crímenes, con su validación correspondiente a través de la implementación de un prototipo con productos específicos.

3.3 Estudio de herramientas de Big Data

Dado el interés que está despertando en los últimos años la etapa de procesamiento, resulta interesante hablar de las diferentes alternativas tecnológicas que existen para procesar Big Data. Esta etapa es la responsable de recoger los datos brutos y transformarlos a datos con valor que pueden dar respuesta a la pregunta que nos estamos haciendo. Y para enfrentar esta etapa, se han desarrollado dos paradigmas fundamentales presentados en la figura 6: Batch o en lotes y streaming o tiempo real.

Existen una amplia gama de frameworks open source para resolver un problema de Big Data. Cada una de las soluciones tiene un conjunto diferentes de ventajas, desventajas y aplicaciones ideales. A continuación se definen los principales enfoques de procesamiento, y se comparan herramientas, poniendo énfasis en lo que se necesita para nuestro problema planteado.

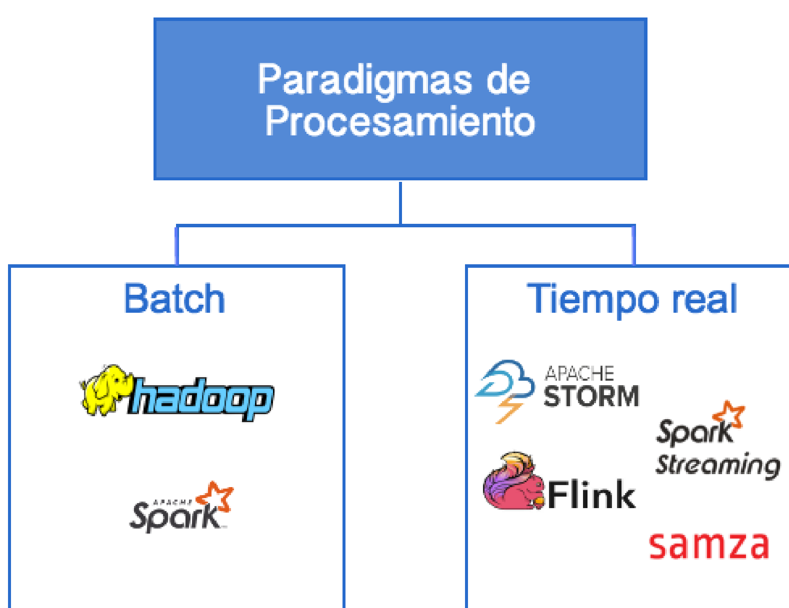


Fig. 6: Paradigmas de procesamiento

3.3.1 Paradigma batch o en lotes

Las primeras tecnologías de Big Data utilizaban el paradigma batch o en lotes. Estos sistemas ejecutan de manera periódica y trabajan con grandes volúmenes de información. El objetivo es acumular la mayor cantidad de datos posible, procesarlos y producir resultados que se agrupan por lotes.
Las principales características de un sistema de procesamiento por lotes son:

- Tener acceso a todos los datos en todo momento.
- Realizar cálculos grandes y complejos.
- Generalmente se enfoca más en el rendimiento que en la latencia de los componentes individuales del cálculo.
- Tiene latencia medidas en minutos o más.

El principal exponente de esta manera de trabajar es Hadoop MapReduce, aunque existen otras menos utilizadas como Apache Pig.
Hay algunos escenarios en los que se recomienda la utilización de este tipo de herramientas, como por ejemplo cuando no se necesita un cálculo con una periodicidad alta, cálculos que solo se deben ejecutar a fin de mes (por ejemplo facturas de una organización), generación de informes con una periodicidad baja, entre otros.

3.3.1.1 Hadoop

Hadoop [24] un sistema de código abierto que se utiliza para almacenar, procesar y analizar grandes volúmenes de datos, en el orden de los petaBytes o más.
Surgió como una iniciativa open source a partir de las publicaciones de artículos por parte de Google de sus sistemas de archivos, su herramienta de mapa y el sistema BigTable Reduce. Como consecuencia, se generaron un conjunto de soluciones en el entorno Apache entre las que se encuentran HDSF Apache, MapReduce, HBase y las cuales en conjunto se conocen como Hadoop.

En la actualidad los sistemas que utilizan las organizaciones son capaces no solo de generar un gran volumen de datos estructurados (SQL), sino que también generar una cantidad importante de datos no estructurados (NoSQL). Hadoop es capaz de almacenar todo tipo de datos: estructurados, no estructurados, semi estructurados, archivos de registro, audio, video, entre otros. Por otra parte también se destaca la capacidad de brindar alta disponibilidad y recuperación de datos.

Entre los componentes de Hadoop se encuentran:

- HDSF: consiste en un sistema de archivos distribuido, capaz de distribuir la información a distintos dispositivos en vez de guardarla en una sola máquina.
- MapReduce: es un framework que permite aislar a los programadores de todas las tareas propias de la programación en paralelo. En otras palabras permite que un programa escrito en los lenguajes comunes, se pueda ejecutar en un conjunto de máquinas (cluster de Hadoop).
- YARN (Yet Another Resource Negotiator): asigna CPU, memoria y almacenamiento a las aplicaciones que se ejecutan en un cluster Hadoop. Las primeras distribuciones de Hadoop solo permitían ejecutar aplicaciones MapReduce, pero con YARN se logra que otros frameworks de aplicaciones también puedan ejecutarlo.

A continuación se profundizará en estos componentes principales de Hadoop:

HDSF

Hadoop Distributed File System (HDSF) es un sistema de archivos distribuidos diseñado para ejecutarse en hardware básico. Tiene muchas cosas en común con los sistemas de archivos distribuidos ya existentes. Sin embargo, tiene diferencias significativas. HDSF es tolerante a fallas, proporciona acceso de alto rendimiento a los datos de la aplicación y es adecuado para aplicaciones que tienen un conjunto grande de datos.
Tiene una arquitectura maestro-esclavo. Un cluster de HDFS consiste en un único nodo llamado NameNode, un servidor maestro encargado de gestionar el espacio de nombres del sistema de archivos y el acceso a los mismos por parte de clientes. A su vez, existe un DataNode para cada nodo del cluster, los cuales administran el almacenamiento de los nodos en los que se ejecutan. Proporcionan funciones de lectura y escritura, creación, eliminación y replicación.
Un archivo está dividido en bloques que son almacenados en DataNodes.

MapReduce

MapReduce es un paradigma de programación que tiene como objetivo mejorar el procesamiento de grandes volúmenes de datos en sistemas distribuidos, y está especialmente pensado para trabajar con grandes volúmenes de datos (en el orden de los gigabytes o terabytes).
Proporciona una abstracción que oculta la paralelización de cálculo, la distribución de datos y el manejo de los fallos, logrando un alto rendimiento en un cluster.

El nombre MapReduce viene dado por la arquitectura de este modelo de programación, que está dividido principalmente en dos fases que se ejecutan en una infraestructura formada por varios nodos, para formar un sistema distribuido.
El cálculo toma un conjunto de pares clave-valor de entrada, y produce un conjunto de pares clave-valor de salida. El usuario de la librería MapReduce expresa el cálculo como dos funciones: Map y Reduce:

- Map: la función Map toma un par de entrada y produce un conjunto de pares clave-valor intermedio. Se agrupan todos los valores intermedios asociados con la misma clave intermedia y se pasan a la función Reduce, luego de realizado el procesamiento.
- Reduce: la función Reduce, también escrita por el usuario, acepta una clave intermedia I y un conjunto de valores para esa clave. Combina estos valores para formar un conjunto de valores posiblemente más pequeño. Normalmente, solo se produce cero o un valor de salida por invocación de la función Reduce. Los valores intermedios se suministran a la función de reducción del usuario a través de un iterador. Esto nos permite manejar listas de valores que son demasiado grandes para caber en la memoria

Hadoop MapReduce es un framework de software que implementa el paradigma de programación MapReduce, y está pensado para escribir fácilmente aplicaciones que procesan grandes cantidades de datos en paralelo en clusters muy grandes, de manera confiable y tolerante a fallas.
Un job mapReduce divide el conjunto de datos de entrada en trozos independientes, que luego son procesados de forma paralela en las tareas que ejecutan la función map. El framework ordena los resultados de los maps, que luego se ingresan en la tarea de reducción. Normalmente tanto la entrada como la salida de los jobs se almacenan en un sistema de archivos (por ejemplo HDSF). El framework se encarga de programar las tareas, supervisarlas y volver a ejecutarlas cuando fallan.

YARN

La idea fundamental de YARN es dividir las funcionalidades de la gestión de recursos con la de programación y supervisión del trabajo, en demonios separados.
Consiste en tener dos servicios: un ResourceManaager global (RM) y un ApplicationMaster por aplicación (AM), los cuales se encargan de las tareas de gestión de recursos, planificación y monitorización de recursos.
Una aplicación es cualquier job sencillo o un DAG de jobs, siendo un DAG (directed acyclic graph) un grafo dirigido acíclico de jobs.
El ResourceManager y el NodeManager forman el framework de cálculo de datos, donde el ResourceManager es la autoridad máxima que maneja los recursos entre las aplicaciones del sistema. NodeManager es el framework por máquina que es el responsable de monitorear el uso de recursos (CPU, memoria, red, disco) y reportándose con el ResourceManager/Scheduler. El ApplicationMaster por aplicación es una librería específica del framework que tiene la tarea de negociar recursos del ResourceManager y trabajar con los NodeManager para ejecutar y monitorear las tareas.

El ResourceManager se divide en dos componentenes: Scheduler y ApplicationManager.
El Scheduler es el responsable de asignar recursos a las aplicaciones que están ejecutando, sujeto a limitaciones. Es puramente un Scheduler en el sentido de que no realiza monitoreo o seguimiento de estado de las aplicaciones. Además no da garantías sobre el reinicio de tareas fallidas debido a fallas de aplicación o de hardware. Realiza la función de programación en función de los requisitos de recursos de las aplicaciones, lo hace en función de la noción abstracta de un contenedor de recursos, que incorpora elementos como memoria, CPU, red , disco, etc.
El ApplicationManager es responsable de aceptar envíos de trabajos, negociar el primer contenedor para ejecutar el ApplicationMaster específico de la aplicación o proporcionar el servicio para reiniciar el contenedor del ApplicationMaster en caso de falla. El ApplicationMaster para cada aplicación tiene la responsabilidad de negociar contenedores de recursos apropiados desde el Scheduler, hacer un seguimiento de su estado y monitorear su progreso.

Arquitectura de YARN:
En la figura 7 se muestra un diagrama de la arquitectura de YARN para un mejor entendimiento de la misma. El ResourceManager y el NodeManager esclavo de cada nodo forman el framework de trabajo, como se dijo anteriormente, el ResourceManager se encarga de repartir y gestionar los recursos entre todas las aplicaciones del sistema, mientras que el ApplicationManager se encarga de la negociación de los recursos con el ResourceManager y el NodeManager. Estos recursos los utiliza para poder ejecutar y controlar las tareas. [25]

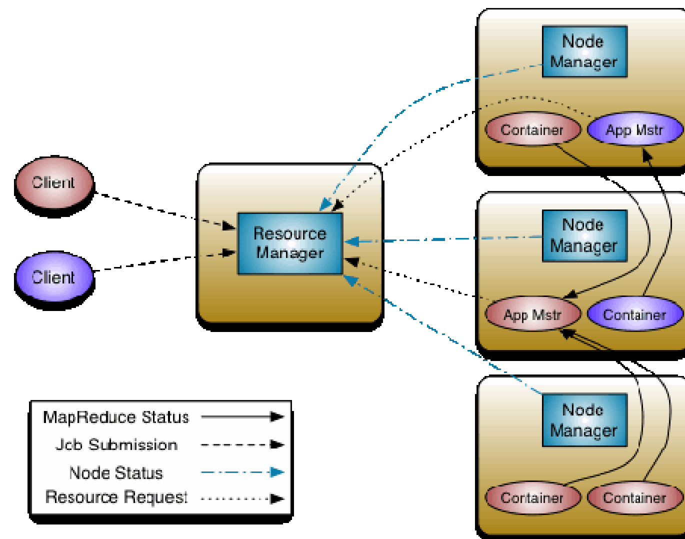


Fig. 7: Arquitectura Apache YARN

3.3.2 Paradigma en tiempo real

Los sistemas en tiempo real o streaming procesan los datos ni bien entran al sistema, y lo hacen de manera continua. Esto permite que en tiempos pequeños se procesen de manera analítica la mayoría de los datos.
A la hora de implementar un sistema de este tipo se deben tener en cuenta ciertos aspectos para su correcto funcionamiento. Se debe disponer de suficiente memoria para almacenar los datos en por ejemplo colas de mensajes, así como también la capacidad de procesamiento debería ser igual o más rápida que la entrada de datos.
En el problema planteado en este proyecto, se tiene la necesidad de una solución en tiempo real que permita la toma de decisiones y alertas al instante. A su vez se consideran herramientas open source, con baja latencia, que permitan construir sistemas distribuidos, escalables y tolerante a fallas.

Se realiza previamente una investigación de las herramientas existentes en el área de Big Data con habilidad de procesamiento en tiempo real, y se finaliza seleccionando cuatro de ellas para hacer un análisis más exhaustivo.

3.3.2.1 Spark

Apache Spark es un framework open source para el procesamiento de datos masivos diseñado para ser veloz, fácil de utilizar, y con capacidades avanzadas de analítica. Se puede ejecutar en clusters de Hadoop a través del modo independiente YARN o Spark. Puede procesar datos en HDFS, HBase, Cassandra, Hive y cualquier formato de entrada Hadoop. Está diseñado para realizar el procesamiento por lotes (similar a MapReduce) y nuevas cargas de trabajo como streaming de datos, consultas interactivas y Machine Learning.

Entre sus plataformas se encuentran:

- Spark SQL: permite consultar datos estructurados usando SQL, o una API que se puede usar desde Java, Scala o Python.
- Spark Streaming: ofrece además del procesamiento en lotes, la posibilidad de gestionar datos en tiempo real. Esto agiliza el procesamiento de los datos ya que se analizan según se van generando, sin ningún tiempo de latencia.
- MLlib (Machine learning): contiene algoritmos que proveen a Apache Spark de muchas utilidades, que incluyen clasificación, regresión, clustering, filtrado colaborativo, reducción de dimensionalidad, entre otros.
- GraphX: es un framework de procesamiento gráfico el cual proporciona una API para elaborar grafos a partir de los datos. Al comienzo fue un proyecto separado, uniéndose posteriormente al proyecto Apache Spark.

Cada aplicación Spark consiste en un programa de controlador que ejecuta la función principal del usuario y ejecuta varias operaciones paralelas en un cluster.
La abstracción principal que proporciona Spark es un conjunto de datos distribuidos resiliente (RDD), la cual es una colección de elementos divididos en los nodos del cluster que pueden operarse en paralelo. Los RDD se crean comenzando con un archivo, en el sistema de archivos de Hadoop (o cualquier otro sistema de archivos compatible con Hadoop) o una colección de Scala existente en el programa del controlador, y transformándolo. Los usuarios también pueden pedirle a Spark que conserve un RDD en la memoria, permitiendo que se reutilice de manera eficiente en operaciones paralelas. Los RDD se recuperan automáticamente de las fallas de los nodos. [26]

Spark Streaming es capaz de ingerir datos de varias fuentes, entre los que se incluyen Apache Kafka, Apache Flume, Amazon Kinesis y Twitter, así como sensores y dispositivos conectados por medio de sockets TCP. Asimismo, tiene la habilidad de poder procesar datos almacenados en sistemas de archivos HDFS o Amazon S3.

Una de las características más reconocibles de Spark es que se puede decir que es una plataforma de plataformas, que agiliza en funcionamiento y mantenimiento de las soluciones.

3.3.2.2 *Spark Streaming*

Spark streaming es una extensión del core de la API de Spark la cual permite el procesamiento de flujos de datos en tiempo real, con alto rendimiento, escalabilidad y tolerancia a fallas.
Los datos pueden ser ingeridos de muchas fuentes, y pueden procesarse utilizando algoritmos complejos expresados con funciones de alto nivel como map, reduce, join y window. Finalmente, los datos procesados pueden enviarse a sistemas de archivos, bases de datos y dashboards en el momento.

En la figura 8 se puede observar un flujo básico de Spark streaming. En primer lugar recibe los flujos de datos de entrada en tiempo real y los divide en lotes, que luego son procesados por el motor de Spark para generar la secuencia final de resultados en lotes. Esta técnica se denomina micro-batching.

Fig. 8: Flujo Spark streaming

Provee una abstracción de alto nivel llamada DStream que representa un flujo continuo de datos. Internamente, un DStream se representa como una secuencia de RDD. Es decir, Spark streaming agrupa los datos transmitidos en lotes de duración fija (por ejemplo, 1 segundo), y cada lote se representa como un RDD. Una secuencia que no termina de estos RDD se denomina DStream.

La recepción de datos se lleva a cabo por el receptor, el cual recibe datos y los almacena en Spark. El procesamiento de datos transfiere los datos almacenados en Spark a DStream. Luego se pueden aplicar las dos operaciones (transformaciones y operaciones de salida) en DStream.

Para una alta disponibilidad en producción, la Spark streaming utiliza ZooKeeper y HDFS. ZooKeeper proporciona almacenamiento de estado y elección de líder. Se pueden iniciar múltiples maestros en su cluster que esté conectado a la misma instancia de ZooKeeper. [27]

3.3.2.3 *Apache Flink*

Apache Flink es un framework de procesamiento de flujos de datos que proporciona la capacidad de distribución de datos, comunicación y tolerancia a fallas.
Esta plataforma fue pensada desde un principio como una alternativa a MapReduce, pero esto no quiere decir que no se pueda acceder o hacer uso de HDSF o YARN.
Este ecosistema se sustenta sobre un núcleo de Flink o Flink Core donde se encuentran todas las APIs y librerías básicas para su funcionamiento. Las APIs principales del núcleo de Flink en las cuales se determinan su comportamiento y el entorno de trabajo de los programas ejecutados son:

- DataSet API: entorno de ejecución en donde se ejecutan las transformaciones de datos provenientes de fuentes estáticas como pueden ser archivos o bases de datos.
- DataStream API: similar a la API de DataSet, pero con la diferencia de que los datos son tomados de fuentes dinámicas, como sockets o colas de mensajes.

Una de las librería principales de esta herramienta es FlinkML, una librería de Machine Learning que provee una gran cantidad de algoritmos.

La arquitectura de Apache Flink funciona de la siguiente manera: cuando se va a ejecutar un proceso, se levanta un JobManager que funciona como coordinador de todo el sistema, y uno o más TaskManager que son los encargados de ejecutar las partes de programa en paralelo.
Cuando se envía un programa al sistema, el optimizador se encarga de transformarlo en un DataFlow el cual es ejecutable de forma paralela por los TaskManager.

Flink ofrece distintos niveles de abstracción a la hora de implementar aplicaciones en tiempo real.

- El nivel de abstracción más bajo ofrece streamings sin estado. Está embebido en la API DataStream a través de una función denominada Process Function. Permite a los usuarios procesar eventos entre uno o más streams, posibilitando a los programas realizar cálculos sofisticados

- En la práctica, la mayoría de las aplicaciones no necesitan la abstracción de bajo nivel descrita anteriormente, sino que se programan contra las API principales, como la API DataStream y la API DataSet. Estas API ofrecen los componentes básicos comunes para el procesamiento de datos, como diversas formas de transformaciones, uniones, agregaciones, ventanas, estados, entre otros, especificados por el usuario. Los tipos de datos procesados en estas API se representan como clases en los respectivos lenguajes de programación.

- La API Table es una DSL declarativa centrada en las tablas. Dicha API sigue el modelo relacional (extendido) en el que las tablas tienen un esquema adjunto (similar a las tablas en bases de datos relacionales) y la API ofrece operaciones comparables, como select , project, union, group by, join, etc.
  Aunque la API Table es extensible por varios tipos de funciones definidas por el usuario, es menos expresiva que las API principales, pero más concisa de usar (menos código para escribir).

- La abstracción de más alto nivel ofrecida por Flink es SQL. Esta abstracción es similar a la API Table tanto en semántica como en expresividad, pero representa programas como expresiones de consulta SQL.

Los programas de Flink se basan en streams y transformaciones. Un stream como se ha mencionado anteriormente, es un flujo de datos mientras que una transformación es una operación que toma uno o más streams como entrada, y produce uno o más flujos de salida.
Cuando se ejecutan estos programas, se asignan a flujos de streamings, que consisten en un conjunto de streams y transformaciones. Cada flujo de datos empieza con uno o más fuentes y termina en uno o más sink. Un sink corresponde al lugar donde salen los streams del sistema, lo que podría representar una base de datos o un conector a otro sistema.
Se puede visualizar el flujo en la figura 9.

Habitualmente hay una correspondencia uno a uno entre las transformaciones en los programas y los operadores en el stream de datos. Sin embargo, algunas veces una transformación puede consistir en operadores de transformación múltiple. [28]

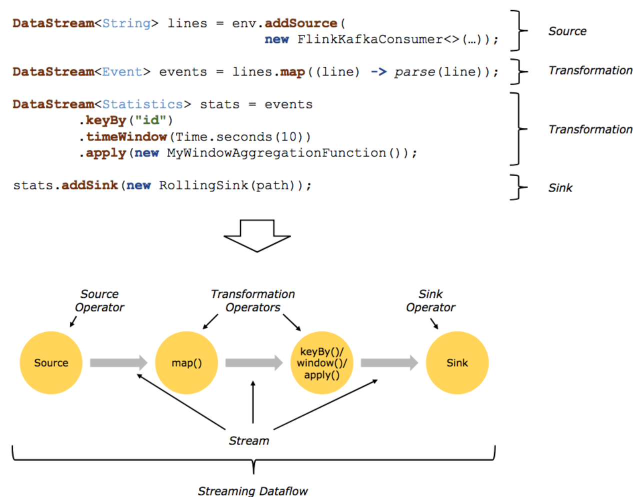


Fig. 9: Flujo Apache Flink

3.3.2.4 Apache Samza

Apache Samza es un framework de procesamiento de streams distribuido. Utiliza Apache Kafka para mensajería, y Apache Hadoop YARN para proporcionar tolerancia a fallas, procesamiento aislado, seguridad y manejo de recursos.

Samza procesa streams, donde un stream está compuesto de mensajes estáticos de un tipo o categoría similar. Por ejemplo, un stream podría ser todos los clicks en un sitio web, las actualizaciones de una tabla de una base de datos, los logs producidos por un servicio, o cualquier otro tipo de datos producidos por un evento particular. Los mensajes pueden ser agregados a un stream o ser leídos desde un stream. Puede tener cualquier número de consumidores; la lectura de una secuencia no elimina el mensaje. Los mensajes pueden tener opcionalmente una clave asociada que se usa para particionar.

Entre las características principales se encuentran:

- Manejo de estados: gestión de snapshotting y restauración de estado de procesamiento de un stream. Cuando se reinicia un procesador, Samza restaura su estado a un snapshot consistente. Está diseñado para manejar grandes cantidades de estados.
- Tolerancia a fallas: cada vez que falla una máquina en un cluster, Samza trabaja con YARN para migrar sus tareas a otra máquina de manera transparente.
- Durabilidad: Utiliza Kafka para garantizar que los mensajes se procesen en el orden en que se escribieron en una partición, y que los mismos no se pierdan.
- Escalabilidad: Samza está particionado y dividido en todos los niveles. Kafka proporciona streamings ordenados, particionados, repetibles y que toleran fallas. YARN proporciona un entorno distribuido para que los contenedores de Samza funcionen.
- Pluggable: Aunque Samza trabaje con Kafka y YARN, también proporciona una API "conectable" que le permite ejecutar en otros sistemas de mensajería y entornos de ejecución.
- Independencia del procesador: Funciona con Apache YARN, que es compatible con el modelo de seguridad de Hadoop.

A continuación se describen los componentes de esta herramienta:
Un job en Samza, representado en la figura 10, es un código que hace una transformación lógica en el conjunto de streams de entrada y agrega un mensaje en el conjunto de streams de salida. Para

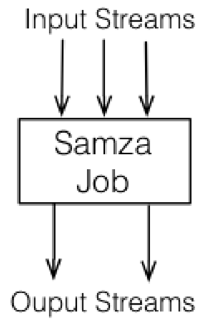


Fig. 10: Representación de un job Samza

escalar en el rendimiento del procesamiento de streams, se dividen los streams y jobs en unidades más pequeñas de paralelismo, particiones y tareas (tasks).

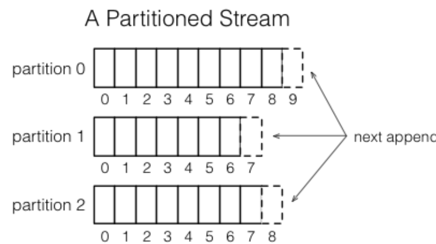


Fig. 11: Representación de un stream Samza

A su vez cada stream se divide en una o más particiones donde cada partición en un stream es una secuencia totalmente ordenada de mensajes. Cada mensaje en la partición tiene un identificador que se llama desplazamiento (offset), que es único dentro de la partición. Cuando se agrega un mensaje a un stream, se agrega a solo una de las particiones del stream. La asignación del mensaje a su partición se hace según una clave elegida cuando se construye. Por ejemplo, si se está procesando datos de distintos usuarios, un id de usuario podría ser una clave y esto garantiza que todos los mensajes relacionados con el mismo usuario terminan en la misma partición.

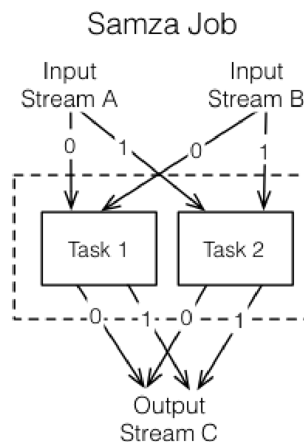


Fig. 12: División de un job Samza en tasks

Un job se escala dividiéndolo en múltiples tasks, como se puede observar en la figura 12. Un task

es la unidad de paralelismo del trabajo, como lo es la partición del stream. Cada task consume datos de una partición para cada uno de los flujos de entrada.
Un task procesa los mensajes de cada una de sus particiones de entrada de manera secuencial, en el orden del desplazamiento de los mensajes, y no existe ordenamiento definido entre las particiones. Esto permite que cada task funcione de manera independiente. El planificador YARN asigna cada una a una máquina, de esta manera el job puede estar distribuido en muchas máquinas.

El número de task en un Job está determinado por el número de particiones (inputs). A su vez, se puede cambiar los recursos asignados a un Job, como pueden ser la cantidad de memoria o el número de procesadores, para satisfacer las necesidades que necesita el Job.
La asignación de una partición a una task nunca cambia, si el task está en una máquina que falla, el task es reiniciado en otra parte y sigue consumiendo de la misma partición. [29]

3.3.2.5 Apache Storm

Apache Storm es un sistema de computación en tiempo real distribuido, y open source. Facilita el procesamiento de flujos de datos ilimitados, haciendo para el procesamiento en tiempo real lo que Hadoop hizo para el procesamiento en lotes. Puede ser utilizado con cualquier lenguaje de programación, y es escalable, tolerante a fallos y fácil de utilizar.

La lógica de una aplicación en tiempo real está contenida en una topología de Storm. Una topología es análoga a un job de MapReduce, y se ejecuta indefinidamente hasta que una persona la termine. Es un grafo de spouts y bolts que están conectadas con agrupamientos de los streams.

Un stream es una secuencia ilimitada de tuplas que se procesan y se crean en paralelo de forma distribuida. Dichas tuplas pueden contener los tipos básicos y otros personalizados. Se pueden enviar objetos en las tuplas.

Un Spout es una fuente de streams en una topología. Éstos leen tuplas de una fuente externa (Twitter, Web Service, colas de mensajes, entre otros) y las emiten hacia dentro de la topología. Los Spouts pueden ser confiables o no confiables. Un Spout confiable puede reenviar una tupla si Storm no pudo procesarla, mientras que un Spout no confiable se despreocupa de la tupla luego que la emite. Para proporcionar la confiabilidad de los Spouts, Storm proporciona dos métodos que son Ack y fail. Ack se invoca cuando se detecta que una tupla emitida se proceso exitosamente a través de una topología, y fail se invoca en el caso contrario.
Cada Spout puede emitir más de un stream, para hacer esto se tiene que declarar varios streams de salida, a través de los métodos correspondientes.

Un Bolt se encarga del filtrado, las funciones, agregaciones, joins, conexiones con base de datos, entre otros, realizando todo el procesamiento. De la misma manera que los spouts, los bolts también pueden emitir más de un stream.

Agrupamiento de Streams: Parte de la definición de la topología es especificar para cada Bolt qué stream debe recibir como entrada. En Storm hay ocho agrupaciones de streams integradas, y se puede implementar una agrupación personalizada implementando la interfaz CustomStreamGrouping. Esto vuelve más potente a esta herramienta, pudiendo adecuarse a las necesidades que se tenga.

Los distintos agrupamientos son :

- Shuffle grouping: las tuplas se distribuyen de manera aleatoria a través de las task de los bolts, de manera que cada bolt tenga la misma cantidad de tuplas.
- Fields grouping: el stream se divide por los campos especificados en la agrupación. Por ejemplo si la tupla tiene un IdUsuario y se agrupa por el mismo, entonces todas las tuplas

que tengan el mismo idUsuario van a ir al mismo task; mientras que las tuplas con distintos idUsuario pueden ir a diferentes task.

- Partial key grouping: Los streams se particionan de acuerdo a los campos especificados en el grouping, pero la carga se balancea entre dos bolts. Esto proporciona una mejor utilización de los recursos en escenarios en donde los datos entrantes estén sesgados, es decir, que la partición por la clave (key) haga que hayan bolts que reciban más tuplas que otros.
- All grouping: el stream se replica en todos los bolts de una task. En la documentación oficial de Storm advierten de tener cuidado al utilizar esta agrupación.
- Global grouping: todo el stream va a una task del bolt. Específicamente va a la task con id más bajo.
- None grouping: esta agrupación específica que no importa cómo se agrupa el stream. Esta agrupación es equivalente a shuffle grouping.
- Direct grouping: un stream agrupado de esta manera implica que el productor del stream decide qué task del consumidor recibirá la tupla.
- Local or shuffle grouping: Si el bolt de destino tiene una o más task en el worker process, entonces las tuplas se envían solo a las task del proceso. De lo contrario actúa como un shuffle grouping.

Paralelismo de Storm: Hay tres entidades principales que se utilizan para ejecutar una topología en un cluster Storm: Worker process, Executors y Tasks.

Un worker ejecuta un subconjunto de una topología, y puede ejecutar uno o más Executors para uno o más componentes (spouts o bolts). Por lo tanto una topología en ejecución consiste en muchos de estos workers que se ejecutan en muchas máquinas dentro de un cluster.

Un Executor es un thread (hilo) generado por un worker process, puede ejecutar una o más task para el mismo componente(spout o bolt).

Una task es la que realmente realiza el procesamiento de los datos. Cada componente que se implemente ejecuta la cantidad de task que se le definen a través del cluster. El número de task para un componente es siempre el mismo durante la vida de una topología, pero la cantidad de executors para un componente puede cambiar con el tiempo. Por lo tanto la siguiente condición siempre es verdadera : $\#threads \leq \#task$. De forma predeterminada el número de tareas se establece para que sea igual al número de ejecutores, es decir, por defecto Storm ejecuta una tarea por hilo.

Arquitectura de un cluster Storm: La arquitectura de un cluster Storm se describe en la figura 13.

- Nimbus: Storm tiene un nodo maestro llamado Nimbus que administra las topologías en ejecución. Éste acepta los request para implementar una topología en Storm y asigna a los workers de todo el cluster para ejecutar esa topología. Nimbus también es responsable de detectar cuándo mueren los workers y reasignarlos a otras máquinas cuando sea necesario.
- ZooKeeper: ZooKeeper es otro proyecto de Apache que se destaca por mantener pequeñas cantidades de estado y tiene una semántica adecuada para la coordinación del cluster. En una arquitectura Storm, ZooKeeper rastrea dónde se asignan los workers y otra información de configuración de topología. Un cluster típico de ZooKeeper para Storm es de tres o cinco nodos.
- Worker nodes: El último grupo de nodos en un cluster Storm comprende sus nodos workers. Cada nodo ejecuta un demonio llamado Supervisor que se comunica con Nimbus a través de ZooKeeper para determinar qué se debe ejecutar en la máquina. El Supervisor luego inicia o detiene los procesos del worker según sea necesario, según lo indicado por Nimbus. Una vez que se ejecutan, los procesos de los workers descubren la ubicación de otros workers a través de ZooKeeper y se pasan mensajes entre ellos directamente.

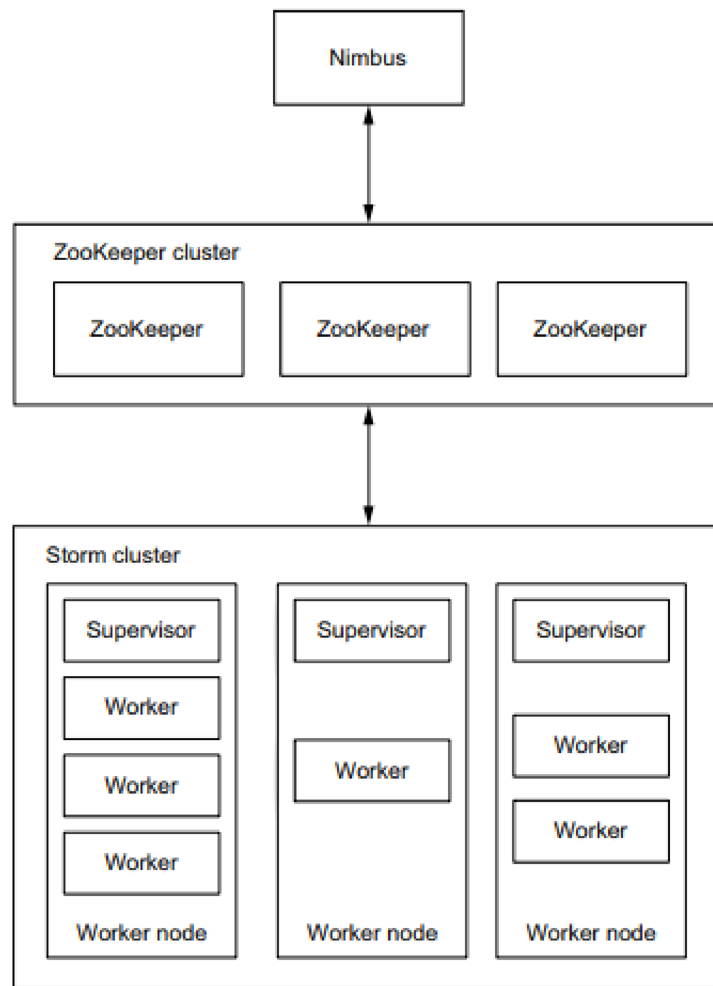


Fig. 13: Arquitectura Apache Storm

Storm garantiza que toda tupla del spout va a ser completamente procesada por la topología. La abstracción básica de esta herramienta provee una garantía de procesamiento de at-last-once, esto quiere decir que garantiza que los mensajes sean procesadas al menos una vez. Los mensajes son reenviados sólo cuando ocurren fallas.

Fue diseñado desde un principio para ser usado con cualquier lenguaje de programación. En el core de Storm hay una definición Thrift para definir y ejecutar topologías. De manera similar los spouts y bolts pueden ser definidas en cualquier lenguaje.

Es tolerante a fallas, es decir cuando un worker falla, Storm automáticamente lo reinicia. Si un nodo falla, el worker es restaurado en otro nodo.
El demonio Nimbus de Storm y el supervisor están diseñados para ser sin estados y fail-fast, por lo tanto, si uno falla son reiniciados como si nada hubiera pasado.

Trident: Es una extensión de Storm. Provee un nivel de abstracción mayor junto con procesamiento de stream con estado y consultas distribuidas de baja latencia.
Utiliza spouts y bolts, pero son auto generados por Trident antes de la ejecución. Posee funciones, filtros, joins, agrupamientos y agregaciones.

Trident procesa los streams como series de batch los cuales se llaman transacciones. Dependiendo del stream de entrada, estos batch serán del orden de miles de millones de tuplas. Aquí se encuen-

tra una de las diferencias con Storm, ya que Storm procesa una tupla por vez.
Por lo tanto resulta útil usar Trident cuando el requerimiento sea procesar solo una vez.
Un fallo en el procesamiento de un batch hace que toda la transacción se retransmita. Para cada batch, Trident llama a la función beginCommit al comienzo, y commit al final de la transacción.

La API de Trident expone la clase "Trident Topology", la cual recibe la entrada de un spout y realiza una secuencia de operaciones en el stream. La tupla de Storm (Storm Tuple) se reemplaza por una tupla de Trident (Trident Tuple), y los bolts se sustituyen por operaciones.

Proporciona un mecanismo para mantener estados, donde los datos se pueden almacenar en la topología o en una base de datos separadas. Esto implica que si una tupla falla durante el procesamiento, entonces esta tupla es retransmitida. [30] [7]

3.3.2.6 Apache Kafka

Para entender el funcionamiento de Kafka es útil comenzar por saber qué es un sistema de mensajes.
Un sistema de mensajería es responsable de transferir datos de una aplicación a otra, por lo que las aplicaciones pueden enfocarse en los datos, pero no preocuparse por cómo compartirlos. La mensajería distribuida se basa en el concepto de colas de mensajes confiables. Los mensajes se ponen en cola de forma asincrónica entre las aplicaciones cliente y el sistema de mensajería. Hay dos tipos de patrones de mensajes disponibles: uno es punto a punto y el otro es el sistema de mensajería de publicación-suscripción (pub-sub). La mayoría de los patrones de mensajes siguen a pub-sub.

- Punto a punto: En un sistema punto a punto, los mensajes se conservan en una cola. Uno o más consumidores pueden consumirlos, pero un mensaje en particular puede ser consumido por un máximo de un solo consumidor. Una vez que un consumidor lee un mensaje en la cola, desaparece de esa cola. El ejemplo típico de este sistema es un Sistema de procesamiento de pedidos, donde cada pedido será procesado por un Procesador de pedidos, pero los Procesadores de pedidos múltiples también pueden funcionar al mismo tiempo.
- Publicación-Suscripción: En este sistema los mensajes se conservan en un topic. A diferencia del sistema punto a punto, los consumidores pueden suscribirse a uno o más topics y consumir todos los mensajes en ese topic. En el sistema pub-sub, los productores de mensajes se denominan editores y los consumidores de mensajes se llaman suscriptores. Un ejemplo de la vida real es Dish TV, que publica diferentes canales como deportes, películas, música, etc., y cualquiera puede suscribirse a su propio conjunto de canales y obtenerlos siempre que sus canales suscritos estén disponibles.

Apache Kafka es un sistema de mensajería de publicación y suscripción distribuido, y una cola robusta que puede manejar un gran volumen de datos y le permite pasar mensajes de un punto final a otro. Kafka es adecuado para el consumo de mensajes en línea y fuera de línea. Se basa en el servicio de sincronización de ZooKeeper.
Dentro de los beneficios se pueden encontrar la escalabilidad, confiabilidad dado que se distribuye, replica y es tolerante a fallas, la durabilidad ya que los mensajes se persisten en disco lo más rápido posible y se replican dentro del cluster para evitar la pérdida de datos. Y por último el alto rendimiento tanto para la publicación como la subscripción. Cabe destacar que Kafka mantiene su rendimiento estable incluso cuando se almacenan muchos terabytes de mensajes, lo cual en un problema de Big Data es crucial.
Permite programar productores/consumidores en diferentes lenguajes: Java, Scala, Python, Ruby, C++, entre otros. Está escrito en Scala y es creado por LinkedIn.

Una instancia en un cluster de Kafka se llama Broker. Se puede acceder a todo el cluster, conectándose a cualquier broker, el cual se llama servidor de arranque. Cada broker se identifica mediante un id numérico en el cluster. Para iniciar un cluster de Kafka, tres brokers es un buen número,

pero hay grupos con cientos de brokers.

Un Topic es una categoría o nombre lógico al que se publican los registros. Los topics son siempre múltiples suscriptores; es decir, un topic puede tener cero, uno o muchos consumidores que se suscriban a los datos escritos en él.
Para cada topic, el cluster Kafka mantiene un registro particionado, como se puede visualizar en la figura 14.

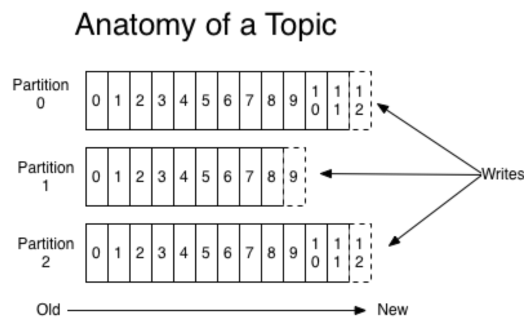


Fig. 14: Anatomía de un topic Storm

Kafka dispone de cuatro APIs principales:

- Producer API: permite a los clientes conectarse a los servidores de Kafka que se ejecutan en el cluster y publicar la secuencia de registros en uno o más topics de Kafka.
- Consumer API: permite a los clientes conectarse a los servidores de Kafka que se ejecutan en el cluster y consumir streams de registros de uno o más topics de Kafka.
- Streams API: permite a los clientes actuar como procesadores de streams al consumirlos de uno o más topics y producir los streams para otros topics de salida.
- Connector API: permite escribir código de productor y consumidor reutilizable

Por lo tanto, Kafka es simplemente una colección de topics que se separan en una o más particiones. Una partición es una secuencia lineal y ordenada de mensajes, donde cada mensaje se identifica por el offset. [31]

El conjunto de pasos que sigue Kafka es la siguiente:

- Los Producers envían mensajes a un topic en intervalos regulares.
- El Broker de Kafka almacena todos los mensajes en las particiones configuradas para ese topic. Asegura de que los mensajes son compartidos en cantidades iguales en las particiones.
- El consumer se subscribe a un topic específico.
- Una vez el consumer se subscribe al topic, Kafka provee al consumidor el id (offset) del topic y lo guarda en ZooKeeper.
- El consumidor pedirá a Kafka nuevos mensajes en intervalos regulares.
- Una vez que Kafka recibe mensajes de los producers, los envía a los consumers.
- El consumer recibe el mensaje y lo procesa.
- Cuando los mensajes son procesador, el consumer manda mensajes de ACK al broker de Kafka.

- Al momento que Kafka recibe el mensaje de ACK, modifica el offset a un nuevo valor y lo actualiza en ZooKeeper.
- Este flujo continuará hasta que el consumer pare el request.

ZooKeeper

ZooKeeper es un servicio centralizado encargado de administrar y gestionar la coordinación entre procesos en sistemas distribuidos.
Expone un conjunto simple de primitivas sobre las que las aplicaciones distribuidas pueden construir para implementar servicios de nivel superior para la sincronización, el mantenimiento de la configuración y los grupos y nombres. Está diseñado para ser fácil de programar y utiliza un modelo de datos diseñado según la estructura de árbol de directorios familiar de los sistemas de archivos. Se ejecuta en Java y se puede integrar tanto con Java como con C.

ZooKeeper permite que los procesos se coordinen entre sí a través de un espacio de nombres compartido de estructura jerárquica, parecido a un sistema de archivos, donde cada nodo recibe el nombre de znode. Los znodes son similares a los de un sistema de archivos normal y corriente, y cada uno tiene un padre que es el znode anterior. Los znodes tampoco pueden ser eliminados si tienen hijos, tal y como pasa con los sistemas de archivos comunes.

Para asegurarse de que tenga alta disponibilidad los servicios están replicados a distintos nodos del cluster (se convierten en servidores ZooKeeper) donde está instalado, y se mantienen sincronizados a través de logs de transacciones e imágenes de estado en un almacenamiento permanente (como por ejemplo una base de datos). Un proceso cliente del servicio ZooKeeper se conecta únicamente a uno de los servidores, obteniendo de él una clave de sesión y desde el que realiza todas las peticiones a través de una conexión TCP. En caso de que la conexión se pierda, el cliente trata de conectarse a otro de los servidores, renovando la sesión. [32] [21]

3.4 Comparación de herramientas de Big Data en tiempo real

Al comparar las distintas herramientas para trabajar con Big Data se deben tener en cuenta varios aspectos de las mismas, entre los que se encuentran:

- Procesamiento: se considera si el procesamiento de datos es en tiempo real (stream) o en lotes (batch).
- Ordenamiento y garantías: implica cuál es la garantía de que ante cualquier eventualidad se procese un registro entrante en un motor de streaming. Se identifican tres tipos:
  - at-most-once: no se puede procesar en caso de fallas.
  - at-least-once: se procesará al menos una vez incluso cuando existan fallas.
  - exactly-once: se procesará exactamente una vez.
- Administración de estados: en el caso que se requiera procesamiento donde se necesita mantener algún estado, las herramientas deberían proporcionar algún mecanismo para preservar y actualizar la información de dicho estado.
- Particionamiento y paralelismo: interesa conocer las técnicas de reparto y carga de datos, así como también el paralelismo.
- Despliegue y ejecución
- Latencia y buffering: la latencia debe ser lo más baja posible mientras que el rendimiento debe ser lo más alto posible. Aunque es difícil lograr ambos al mismo tiempo, se debe buscar un equilibrio.
- Interoperabilidad: dado que nos basamos en el desarrollo de la capa de ingestión, resulta necesario conocer con qué otras herramientas o sistemas pueden integrarse.
- Lenguajes soportados: basada la experiencia de los integrantes del grupo y el alcance del proyecto, interesa considerar los lenguajes que permite utilizar cada herramienta para su desarrollo.
- Comunidad: es necesario contar con una fuerte comunidad detrás, para evitar posibles complicaciones y atrasos.

Asimismo no se tiene que perder de vista las necesidades inherentes a la solución que se quiere implementar, las cuales derivan de los requerimientos [33] [34].

3.4.1 Ordenamiento y garantías

Apache Storm permite que el usuario pueda elegir el nivel de garantía de procesamiento de mensajes que quiere utilizar. El modelo más simple at-most-once (como máximo una vez) descarta los mensajes si no son procesados correctamente. No se requiere una lógica especial, sino que procesa los mensajes en el orden en el que llegan desde el spout.
También está el modelo at-least-once (al menos una vez) el cual garantiza que los mensajes sean procesados al menos una vez.
Por último ofrece el modelo exactly-once (exactamente una vez) usando su abstracción Trident. Este modelo usa el mismo mecanismo de detección de at-least-once, y a su vez, los estados de Storm permiten la detección de mensajes duplicados para ignorarlos.

Por otra parte, Samza ofrece solo la garantía de procesamiento at-least-once. Procesa siempre los mensajes en el orden en el que aparecen en la partición, pero no hay garantías de orden entre diferentes streams de entrada o particiones. En Samza no se ven ventajas en el uso del modelo at-most-once (en el que se pueden perder mensajes), por eso es que no se implementa. Con esta herramienta el procesamiento de los mensajes está garantido.

Spark streaming garantiza el procesamiento ordenado de los RDD en un DStream. Pero como cada RDD se procesa en paralelo, no hay orden garantizado dentro del RDD. Este es un diseño realizado por Spark.

Si se desea procesar los mensajes en orden dentro del RDD, se tienen que procesar en un hilo, lo que provoca que se pierdan los beneficios de la paralelización. Si hablamos de garantías, Spark streaming garantiza el modelo exactly-once.

Flink utiliza el modelo exactly-once para cálculos con estado. Esto significa que las aplicaciones pueden mantener una agregación o resumen de los datos que se han procesado a lo largo del tiempo, y el mecanismo de punto de control de Flink, asegura la semántica exactamente una vez para el estado de una aplicación en caso de falla.

3.4.2 Administración de estados

La API de menor abstracción para los Bolts de Storm no ofrece ninguna ayuda para el manejo de estado de los mensajes en el procesamiento de los streams. Un bolt podría guardar en memoria un estado (el cual se perdería si el bolt se muere), o podría hacer una llamada a una base de datos remota para leer o guardar estados. De cualquier manera, la topología procesa mensajes a una velocidad mucho superior que las llamadas a una base de datos remota, por lo tanto, esta llamada se convertiría rápidamente en un cuello de botella.
Como parte de su API de alto nivel Trident, Storm ofrece una administración de estados automática. Mantiene los estados en memoria y periódicamente guarda puntos de control (checkpoints) en una base de datos remota para tener durabilidad. Al mantener metadata junto con los estados, Trident es capaz de lograr la semántica de procesamiento de exactly-once. Este enfoque de Storm de guardar los estados funciona bien si la cantidad de estados por cada bolt es bastante pequeña (inferior a 100 KB). Si se necesita mantener una cantidad grande de mensajes este enfoque se degrada.

Samza utiliza otro enfoque completamente distinto para la administración de estados. En lugar de utilizar una base de datos para el almacenamiento en el tiempo, cada task de Samza incluye un almacenamiento clave-valor ubicado en la misma máquina. En comparación con la conexión a través de la red a una base de datos remota, el estado local de Samza permite leer y escribir grandes cantidades de datos con un mejor rendimiento. Samza replica este estado en múltiples máquinas para manejar la tolerancia a fallas.
La limitación que tiene el manejo de estados en dicha herramienta es que al momento no admite la semántica de procesamiento exactly-once, pero es un punto en el cual los desarrolladores están interesados y están trabajando.
Las tareas pueden almacenar y consultar datos a través de las API proporcionadas por Samza.

Spark streaming ofrece un DStream de estado, el cual mantiene el estado para cada clave y una operación de transformación llamada updateStateByKey para modificarlo.
Cada vez que se ejecuta esta operación, obtendrá un nuevo estado de DStram el cual se actualiza al aplicar la función pasada a updateStateByKey. Esta transformación puede funcionar como un almacenamiento clave-valor básico, aunque tiene algunos inconvenientes.
Spark Streaming escribe periódicamente datos intermedios de operaciones con estado en HDFS. En el caso de updateStateByKey, el RDD de estado completo se escribe en el HDFS después de cada intervalo de punto de control.

En el caso de Flink, provee un mecanismo de tolerancia a fallas basado en puntos de control (checkpoints), que son capturas instantáneas automáticas y asíncronas del estado de una aplicación. En caso de fallas, un programa Flink que tenga habilitados los puntos de control, recuperará el procesamiento en el último punto de control completado.
A su vez también incluye un mecanismo llamado savepoints, que son puntos de control activados manualmente. Esto permite que un usuario pueda generar un savepoint, detener el programa en ejecución y luego reanudarlo desde el mismo estado y posición del stream.

3.4.3 Particionamiento y paralelismo

El modelo de paralelismo de Samza y Storm son muy similares. Ambos frameworks separan el procesamiento en tasks independientes que corren en paralelo. La asignación de recursos es independiente al número de task dado que un job pequeño puede mantener todas las tasks en un solo procesador, y un job grande puede distribuirlas en muchos procesadores en muchas máquinas.

La mayor diferencia radica en que Storm usa por defecto un hilo por task, mientras que Samza usa procesos de un solo hilo (contenedores). Un contenedor Samza puede contener múltiples tasks, pero solo hay un hilo que invoca cada una de ellas. Esto significa que cada contenedor se asigna a exactamente un núcleo de CPU, lo que hace que el modelo de recursos sea mucho más simple y reduce la interferencia de otras tasks que se ejecutan en la misma máquina. El modelo multiproceso de Storm aprovecha mejor el exceso de capacidad de las máquinas inactivas, a costa de un modelo de recursos menos predecible.
Cuando se usan spouts transaccionales con Trident (necesarios para lograr la semántica de exactly-once), el paralelismo se reduce considerablemente.

El paralelismo en Spark Streaming se logra al dividir el trabajo en pequeñas tareas y enviarlas a los ejecutores. Se tienen dos tipos de paralelismo:

- Paralelismo en la recepción del stream: en este caso una entrada DStream crea un receptor, y éste recibe un stream de datos de entrada y se ejecuta como una tarea de larga ejecución. Para paralelizar esto se puede dividir un stream de entrada en múltiples streams basados en algún criterio. Luego se pueden crear múltiples DStreams para estos streams y los receptores ejecutarán como múltiples tareas.
- Paralelismo en el procesamiento del stream: dado que un DStream es una secuencia continua de RDDs, el paralelismo se puede lograr con operaciones RDD normales como ser map o reduceByKey.

En el caso de Flink, sus programas son paralelos y distribuidos. Durante la ejecución, un stream puede tener varias particiones y cada operador puede tener varias subtareas de operador. Estas subtareas son independientes entre sí y se ejecutan en distintos subprocesos así como también en diferentes máquinas.
El número de subtareas del operador es el paralelismo de ese operador en particular, siendo en un stream el mismo de su operador productor. Diferentes operadores del mismo programa pueden tener diferentes niveles de paralelismo.

3.4.4 Despliegue y ejecución

Un cluster de Storm se compone de un conjunto de nodos que ejecutan un demonio llamado supervisor. Los demonios supervisores se comunican con un único nodo maestro llamado Nimbus, encargado de asignar trabajos y administrar los recursos. Esta operativa es muy similar a YARN, aunque este último es un poco más completo al pretender ser multiframework. Hay muchas similitudes entre el Nimbus de Storm y el ResourceManager de YARN, así como también entre los supervisores de Storm y los nodeManagers de YARN.

A su vez la idea de Samza es directamente usar YARN y no tener que escribir su propio framework de gestión de recursos, ya que YARN se caracteriza por ser estable, tiene todas las funcionalidades necesarias y es interoperable con Hadoop. También proporciona otras características interesantes como son seguridad (autenticación por usuario), aislamientos de procesos, entre otros.
El soporte de YARN en Samza es pluggable, esto quiere decir que se puede cambiar el framework de ejecución si se quiere.

En Spark streaming se tiene un objeto SparkContext para comunicarse con los clusters managers, que luego asignan recursos para la aplicación. Actualmente Spark admite tres tipos de cluster managers: Spark standalone, Apache Mesos y YARN.

Flink ofrece dos opciones a la hora de configurar un cluster, una es el standalone cluster (levantando un JobManager y uno o más TaskManagers), y la otra es basado en YARN.

3.4.5 Latencia y buffering

Storm usa ZeroMQ, una librería de mensajería distribuida, para la comunicación entre bolts, lo que permite una transmisión de tuplas con una latencia muy baja. Sin embargo en Samza no hay un mecanismo equivalente y siempre las comunicaciones son a través de streams.

Por otro lado si un bolt produce mensajes más rápido de lo que el consumidor pueda leerlos, el Buffer de zeroMQ en el proceso del bolt que los produce comienza a llenarse. Si este buffer crece demasiado, las tuplas podrían alcanzar el timeout de la topología, lo que hace que los mensajes se vuelvan a emitir y empeora el problema, agregando más mensajes al buffer.
Para evitar este problema se puede configurar la cantidad de mensajes que pueden haber al mismo tiempo en la topología.
La falta de intermediario entre los bolts de Storm agrega complejidad a la hora de tratar con la tolerancia a fallas y la semántica de los mensajes. Storm tiene un mecanismo para detectar las tuplas que no se procesaron, pero Samza no necesita ese mecanismo porque cada flujo de entrada y salida es tolerante a fallas y se replica.
Samza almacena el buffer en disco entre cada pasaje entre task. Esta decisión de diseño permite que el buffer absorba una gran acumulación de mensajes si se retrasa un job en su procesamiento, lo cual implica que tenga una latencia un poco mayor a Storm.

Según el paper de Spark Streaming, con un motor de ejecución rápida puede alcanzar la latencia de tan solo un segundo. En esencia se puede ver a esta herramienta como una secuencia de pequeños procesos batch. Si el procesamiento es más lento que la recepción de datos, se pondrán en cola como un DStream en la memoria, y esta cola irá aumentando. Se aconseja que para lograr una buena transmisión el sistema debe ajustarse hasta que la velocidad de procesamiento sea tan rápida como la recepción.

En Flink de forma predeterminada, los elementos no se transfieren uno a uno a la red sino que se almacenan en el buffer. El tamaño de los buffers se puede configurar en los archivos de configuración de Flink. Si bien este método es bueno para optimizar el rendimiento, puede causar problemas de latencia cuando la transmisión entrante no es lo suficientemente rápida.
Para controlar el rendimiento y la latencia, se puede utilizar el método env.setBufferTimeout (timeoutMillis) en el entorno de ejecución (o en operadores individuales) para establecer un tiempo máximo de espera para que los buffers se llenen. Después de este tiempo, los buffers se envían automáticamente incluso si no están llenos. El valor predeterminado para este tiempo de espera es de 100 ms.

3.4.6 Interoperabilidad

Storm se integra con cualquier sistema de colas y cualquier sistema de base de datos.
Los sistemas de colas como ser Kafka, JMS, Kestrel, son fácilmente integrables gracias a la abstracción del spout de Storm. También lo es la integración con bases de datos ya que se debe abrir una conexión a la misma y realizar operaciones de lectura y escritura. Cuando es necesario, Storm se hace cargo de la paralelización, particionamiento y reintento en caso de fallas.

Si hablamos de Spark Streaming, los datos pueden ser ingeridos de muchas fuentes, como pueden ser Kafka, Kinesis, Twitter o Socket TCP. Además puede recibir datos desde cualquier fuente, más allá de aquellas para las que tiene soporte integrado. Esto requiere que un desarrollador implemente un receptor personalizado para la fuente de datos que se quiere agregar. Hay que tener en cuenta que estos receptores personalizados se pueden implementar solamente con Scala o Java.

Samza lee y escribe mensajes utilizando las interfaces SystemConsumer y SystemProducer. Se puede integrar cualquier intermediario de mensajes con Samza implementando estas dos interfaces.

Flink se integra con muchos otros proyectos en el ecosistema de procesamiento de datos Open source. Funciona con YARN, trabaja con HDSF, y puede ejecutar programas hechos en Hadoop; además se conecta a varios sistemas de almacenamiento de datos.

3.4.7 Lenguajes de programación utilizados

En la siguiente tabla se muestra las distintas herramientas analizadas y los lenguajes que soportan cada una de ellas.

Storm	Spark	Flink	Samza
Todos	Scala, Java, Python	Java, Scala, Python, SQL	Solo lenguajes JVM

Como se ve en la tabla, los componentes de Apache Storm se pueden definir en cualquier lenguaje de programación, lo cual es una gran ventaja con respecto a las otras herramientas que están bastante atadas a ciertas tecnologías particulares (por lo general Java y Scala).

Samza esta escrito en Java y Scala. Se pensó para que soporte multilenguajes, pero actualmente solo soporta lenguajes JVM.

3.4.8 Comunidad

A continuación se realiza un estudio del tamaño de la comunidad de cada herramienta, teniendo en cuenta la cantidad de contribuyentes, cantidad de preguntas en foros, capacidad de respuesta, entre otros.
Creemos que es de gran importancia ya que vamos a utilizar una herramienta que nos es desconocida y precisamos el soporte correspondiente.
El análisis corresponde a datos al día 6 de Noviembre de 2017.

	Spark	Storm	Flink	Samza
Mailing lists	✓	✓	✓	✓
Stack overflow	✓✓✓3443 preguntas (alto % de respuesta)	✓✓2095 preguntas (alto % de respuesta)	✓1348 preguntas	68 preguntas
Git	✓✓✓watch: 1836 star: 14950 fork: 13942 contr: 1161 releases: 51	✓✓watch: 648 star: 4592 fork: 3368 contr: 277 releases: 29	✓✓watch: 369 star: 2871 fork: 1980 contr: 344 releases: 34	✓watch: 47 star: 374 fork: 167 contr: 74 releases: 26
Issue tracker	JIRA	JIRA	JIRA	JIRA

Observamos que todas las herramientas tienen una comunidad fuerte atrás y están en continuo desarrollo. Al ser de código abierto, cuentan con repositorio git y utilizan JIRA como tracker para las issues. A su vez utilizan mailing lists e IRC channels para la comunicación. Todas ellas cuentan con página web oficial y la documentación correspondiente.

Se contempla que Spark es la plataforma con la comunidad más grande, ya que es un proyecto de muchos años y que tiene distintos componentes. Sin embargo, si se toma en cuenta solamente el proyecto de Spark streaming, que es el que se emplearía en el presente proyecto, se observa que tiene una comunidad menor que la de Apache Storm. Por ejemplo en Stack Overflow se tienen 286 respuestas bajo el tag "spark-streaming", pero las mismas tienen un muy alto nivel de respuesta.
Por otro lado, Storm se posiciona como una de las herramientas mejor consolidadas y con un buen soporte de la comunidad, con 2095 respuestas bajo el tag apache-storm y con un alto índice de respuesta.
Le siguen Flink y Samza con una comunidad menor pero que están en continuo crecimiento. Samza es aún pequeña, pero acaba de lanzar la versión 0.7.0. Tiene una comunidad receptiva y se está desarrollando activamente. Es muy utilizada en LinkedIn.

3.5 Conclusión

A partir del análisis realizado anteriormente, se concluye con la siguiente tabla que resume los aspectos principales de cada herramienta, considerando las de procesamiento en tiempo real:

	Spark streaming	**Storm**	**Flink**	**Samza**
Modelo de procesamiento	Micro-batch	Un dato a la vez. Micro-batch con Trident	Un dato a la vez, batch	Un dato a la vez
Ordenamiento y garantías	exactly-once	at-most-once, exactly-once, at-least-once	exactly-once	at-least-once
Lenguajes soportados	Scala, Java, Python (se recomienda Scala)	Cualquiera	Java, Scala	Lenguajes JVM
Comunidad	✓✓	✓✓✓	✓	
Interoperabilidad	✓✓✓	✓✓	✓✓	✓✓
Latencia y Buffering	Alta	Baja	Baja	Media
Administración de estados	Para los DStream	Con Trident	Si	Si

Observamos que todas las herramientas de procesamiento de datos en tiempo real que analizamos son aptas para implementar nuestro caso de estudio.

4 Capítulo 4 - Arquitectura e Implementación

En este capítulo en primer lugar se construye una arquitectura genérica enfocada en el dominio de nuestro problema, y se definen las capas que la componen.
Luego de planteado el diseño genérico, se presenta el diseño de dicha arquitectura con productos específicos, resultantes del estudio comparativo realizado en el capítulo anterior.

4.1 Arquitectura genérica

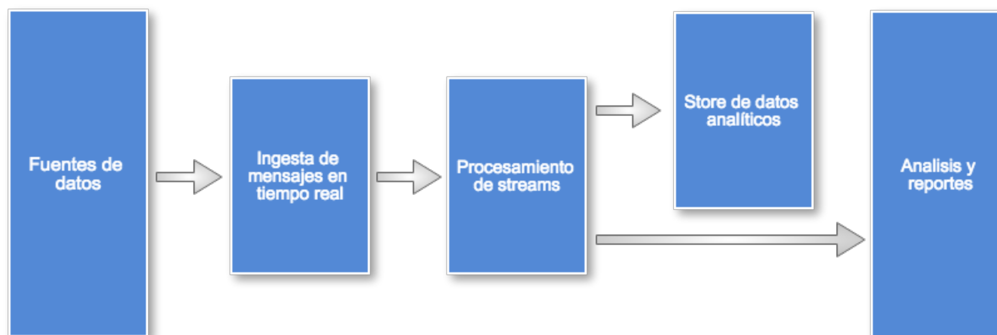


Fig. 15: Diseño Arquitectura genérica

En la figura 15 presentamos el diseño de la arquitectura para el procesamiento de datos en tiempo real, la cual cuenta con los siguientes componentes lógicos:

- Fuentes de datos: todas las soluciones de Big Data comienzan con una o más fuentes de datos, las cuales han crecido en cantidad así como también en distintos formatos en que se presenta.
  Nuestro dominio de trabajo son los datos de crímenes, los cuales pueden proceder de diversas fuentes:
  - Ministerio del Interior: actualmente el Ministerio del Interior cuenta con distintas aplicaciones que tienen como objetivo que los ciudadanos puedan reportar crímenes de una manera rápida, sin tener que acudir a una Seccional Policial. Entre ellas se encuentran la app de 911 y el sistema de denuncias en línea.
    Asimismo, se han instalado cámaras en la ciudad de Montevideo, de parte del Ministerio como también de la Intendencia de Montevideo, las cuales desde su centro de gestión generan alertas de hechos delictivos que se envían a la app de 911. También surgen investigaciones que buscan analizar las imágenes de dichas cámaras a través de su procesamiento, buscando detección de delitos.
    Todo esto implica que se tenga que considerar como posible fuente de datos, la información proveniente de la integración con aplicaciones de esta índole, así como también de sistemas de procesamiento de imágenes.
  - Redes Sociales: el crecimiento de redes sociales y su uso, permite que los ciudadanos realicen quejas o denuncias a través de ellas. Por ejemplo, existen cuentas de Twitter en donde las personas realizan denuncias, por lo que es de interés considerar los datos provenientes de redes sociales.
  - Otros: como ser archivos de texto, planillas, logs, bases de datos.

  Una de las cualidades a considerar es la confiabilidad de los datos ya que las distintas fuentes tienen distintos niveles de confiabilidad. Por ejemplo no es lo mismo la credibilidad de un dato proveniente del 911 que un dato de las redes sociales, por lo tanto hay que asignarle a cada fuente un nivel de confiabilidad/calidad.

- Ingesta de mensajes en tiempo real: dicha arquitectura debe contemplar la captura y almacenamiento de mensajes en tiempo real, para ser consumidos por el procesador de streams. Un caso simple sería implementarlo como un almacén de datos en el que se depositan los

nuevos mensajes en una carpeta, así como también se podría considerar un broker que funcione como buffer para los mensajes. Este intermediario de mensajes debe ser escalable y de entrega confiable.

- Procesamiento de streams: Luego de capturar los mensajes en tiempo real, deben ser procesados por el motor de procesamiento, a través de distintas operaciones como ser filtros, validaciones, agregaciones.
  Este es el centro de la arquitectura, en donde la herramienta que se elija es fundamental para el éxito de la plataforma.
  Entre los puntos que consideramos importantes al elegir una herramienta se encuentran: modelo de procesamiento, baja latencia, alto rendimiento, interoperabilidad, lenguajes soportados y comunidad.

- Store de datos analíticos: se debe considerar el almacenamiento de datos tanto estrcturados como no estructurados, para que puedan ser consultados utilizando herramientas de análisis.

- Análisis y reportes: es de interés para el usuario final contar con algún medio de visualización de los datos, por lo que se debe mostrar la información a través de análisis y reportes.

4.2 Arquitectura con productos específicos

Basándonos en el estudio y comparación de herramientas desarrollado en el capítulo 3, y considerando el diseño de la arquitectura presentado anteriormente, decidimos que la herramienta que mejor se adapta y con la cual creemos que obtendremos mejores resultados es Apache Storm.

Apache Storm ofrece el modelo de procesamiento en tiempo real requerido, soporta una mayor cantidad de lenguajes de programación, posee una comunidad grande la cual puede ayudarnos en los distintos problemas que nos encontremos durante el desarrollo de la plataforma, y a su vez ofrece baja latencia.
Asimismo es la herramienta que se decide utilizar en el artículo de referencia [2].
Storm es una herramienta consolidada, si se quiere buscar algún ejemplo de utilización o encontrar soluciones a distintos problemas no es difícil encontrar respuestas en Internet (foros, stackoverflow, etc).

Soporta la mayoría de los lenguajes de programación, por lo que si en un futuro se quiere migrar el sistema a otra tecnología se puede llevar a cabo sin tener que cambiar la herramienta.

Algunos motivos por los cuales descartamos el resto de las herramientas son:

- Spark Streaming tiene un modelo de procesamiento en micro-batch y es la que tiene mayor latencia, por lo que no nos es conveniente en un escenario de análisis en tiempo real.
- Flink, a pesar de tener similares características que Storm y según nuestro criterio ser apta para la implementación que queremos llevar adelante, su comunidad sigue siendo pequeña y está en pleno desarrollo por lo que es más difícil encontrar ejemplos y respuestas a los problemas.
- Con Samza sucede lo mismo que Flink, tiene una comunidad en desarrollo y a su vez no ofrece la misma velocidad de procesamiento o garantías que Storm. La única ventaja que tiene es el soporte de estados, que no es una de las características primordiales que se tienen en consideración.

En conclusión, utilizaremos Apache Storm, acompañado de Apache Kafka como cola de mensajes. El poder de procesamiento de datos que soportan estas dos herramientas hacen que se complementen de manera natural. A su vez, Storm es fácilmente escalable gracias a la forma que tiene de consumir datos mediante Spouts.

Finalmente, basados en el estudio previo y los requerimientos definidos, se presenta en la figura 16 la arquitectura de productos específicos en la cual además de mostrar los principales módulos se introducen las herramientas a utilizar.

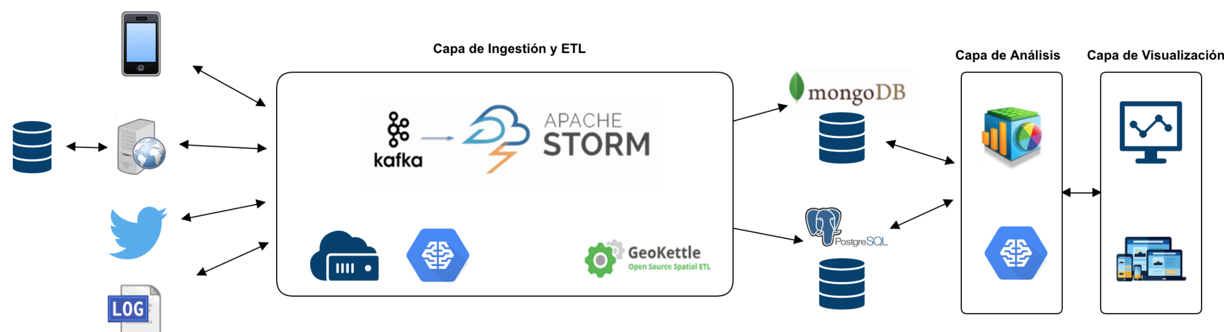


Fig. 16: Diseño Arquitectura basada en productos específicos

A continuación se definen los distintos componentes de la arquitectura:

4.2.1 Entrada de datos

Dado que nos enfocamos en una solución de Big Data, se debe permitir tener variedad de entrada de fuentes de datos, y proveer la capacidad de agregar más. Con esta arquitectura propuesta es posible tener numerosas entradas de datos, como por ejemplo:

- Sistemas de Información a través de Web Services: Es posible obtener datos de los diversos sistemas de información de manejo de crímenes existentes. Dado que no se cuenta con la posibilidad de conectarse directamente a sus bases de datos, esto es viable si se ofrece una API para que puedan comunicarse a través de web services. De esta manera los sistemas de información como por ejemplo el de la policía (911, SGSP), así como también otras aplicaciones de denuncias comunitarias (CityCop), entre otros, logran enviar sus datos a nuestro sistema.

- Redes sociales: dado el gran uso hoy en día, resulta útil la integración con distintas redes sociales, por ejemplo Twitter o Facebook, para obtener información de cuentas de interés.

- Bases de datos: obtener información de distintos motores de bases de datos, por ejemplo puede ser de interés integrarse con bases de datos públicas con datos estadísticos o históricos. Puede ser a tanto con bases de datos relacionales como no relacionales.

- Sensores: hoy en día con la explosión de IoT (Internet de las cosas) se necesita integración con sensores.
  Internet de las cosas (IoT) es un sistema de dispositivos informáticos interrelacionados, máquinas mecánicas y digitales, objetos, animales o personas a los que se proporcionan identificadores únicos y la capacidad de transferir datos a través de una red sin necesidad de de interacciones entre humanos o entre la computadora y un humano.

- Archivos de texto: resulta de utilidad en determinados casos obtener la información de logs de aplicaciones, como por ejemplo el proyecto de grado comentado en el capítulo 2 de analíticas de video.

- Planillas de texto: diversos datos públicos se presentan en planillas Excel por lo que es conveniente proporcionar integración con éstas.

- Servicios geoespaciales: para consultas espaciales, búsqueda de direcciones, lugares de interés, entre otros.

4.2.2 Capa de ingestión y ETL

Esta capa es la encargada de recibir los datos, procesarlos y guardarlos. Para esto se propone utilizar Apache Storm como framework de procesamiento de datos masivos en tiempo real. Aparte de la ingesta, este módulo se encarga de la normalización, validación, clasificación, cruzamientos de datos y la persistencia.

Dado que se tienen variadas fuentes de datos, la etapa de normalización busca uniformizar los datos recibidos y en algunos casos descartar datos inválidos. La información debe ser veraz, por lo tanto hay que validarla; esto se puede realizar de diversas maneras por ejemplo teniendo en cuenta el origen de los datos y la consistencia semántica y sintáctica de los mismos. Estas características se pueden validar con reglas fijas predefinidas. Luego que los datos están validados, interesa clasificarlos agregándoles un atributo de calidad. Este atributo de calidad se define según la fuente de donde proviene el dato, y si existen crímenes similares en el sistema. Para ello se tiene que definir alguna manera de cruzar los datos buscando coincidencias. Al finalizar, los datos se envían a sus respectivas salidas generando alertas y persistiéndolos.

Para la integración con Web Services se propone tener una aplicación que expone un web service que será consumido por las aplicaciones interesadas. Para integrar esta solución con Apache Storm es necesario tener un middleware que los comunique. Se propone Apache Kafka como servidor de cola de mensajes que actúe como intermediario entre estos dos sistemas, donde la aplicación web introduce los mensajes en la cola de Kafka y Storm los consume.

La integración con Twitter se logra a través de la librería twitter4j, que es una librería utilizada para integrar de forma sencilla los servicios de Twitter y la aplicación Java.

Se tiene un módulo de Business Intelligence donde se transforman y cargan los datos a un datawarehouse para su posterior análisis y visualización. Se propone utilizar la herramienta de Pentaho GeoKettle dado que es la que más conocemos, es open source, y hay una gran comunidad detrás.

Dada la cantidad de datos que se generan y el auge que han tenido los últimos años las tecnologías de Inteligencia Artificial, vemos que es muy útil contar con módulos de Machine Learning para proporcionar distintas funcionalidades como pueden ser: clasificación de tweets, validación de datos, identificación de patrones. La continua generación de datos permite entrenar los algoritmos para mejorar su efectividad.

Debido a que los tweets no siempre tienen habilitada la opción de ubicación exacta, puede ser de interés contar con un módulo para que dado el texto del tweet, se identifique el lugar en el que ocurre el crimen, por ejemplo intersección de calles, número de puerta. Para esto se debería validad contra algún servicio externo, como ser GeoNames. GeoNames es una base de datos geográfica gratuita y accesible a través de Internet bajo una licencia Creative Commons.
Provee una API para poder conectarse desde la aplicación Java.

4.2.3 Salida de datos

Se consideran diferentes salidas de datos, teniendo en cuenta las posibles integraciones que puedan surgir en un futuro y que sean de utilidad. Una de ellas es la integración con bases de datos relacionales, en nuestro caso seleccionamos PostgreSQL, dado que el equipo tiene experiencia utilizando dicho motor y se puede fácilmente adicionar la extensión geográfica PostGIS, necesaria para nuestro problema. Esta base de datos se utilizará por ejemplo para realizar la carga del Datawarehouse.

Asimismo resulta interesante integrar la salida de datos con bases de datos no relacionales, dado que estamos ante problemas de variedad de fuentes de datos, las cuales pueden diferir en la estructura que se quiere persistir. Estas bases de datos dan flexibilidad a tener distintas estructuras. En este caso se seleccionó MongoDB, por su facilidad de uso y aprendizaje, y porque es la elegida en Big Data cuando se trabaja con sistemas de gran volumen de datos y heterogéneos.

Dado que estamos ante un sistema en tiempo real, es necesario contar con un conector que se encargue de la comunicación con otras aplicaciones o módulos, y que ésta no sea a través de bases de datos. Para esto, se considera la conexión con otra aplicación a través de web services. En nuestro caso esto resulta provechoso para generar las alertas. Por lo tanto, desde el módulo principal se consumirá un web service que tendrá una definición genérica, el cual se publica en otra aplicación web que será la encargada de mostrar las alertas en un mapa.

En nuestra solución, las siguientes dos capas no son prioridad, dado que el foco se pone en la capa de ingestión y ETL. Sin embargo, utilizamos algunas características de las mismas para la validación del prototipo realizado.

4.2.4 Capa de Análisis

La capa de análisis es la encargada del procesamiento OLAP y algoritmia vinculada a Big Data. Incluye las herramientas de análisis de Business Intelligence.
Consideramos realmente útil contar con un módulo de Machine Learning para permitir retroalimentar y mejorar los resultados de la plataforma. Una aplicación importante es la predicción de crímenes. Gracias a la capacidad de Big Data y teniendo disponible el histórico de datos de distintas fuentes, se puede confeccionar un modelo de predicción de crímenes.

4.2.5 Capa de Visualización

La capa de Visualización ofrece herramientas visuales para la toma de decisiones, como ser dashboards con capacidad geográfica para proveer análisis tanto en tiempo real como general. Contiene sistemas de Monitoreo y de Alertas para agilizar el accionar de los usuarios.

5 Prototipo

En este capítulo se presentan los aspectos más relevantes de la implementación del prototipo, así como también las decisiones tomadas en el desarrollo del mismo. Se explica cada componente utilizando la arquitectura del capítulo anterior, comenzando con las entradas de datos consideradas, la topología de Storm definida, las integraciones con los distintos sistemas externos, decisiones de procesamiento y las salidas de datos.
A su vez se definen las herramientas utilizadas en la implementación, y se presentan las pruebas realizadas para la validación del prototipo.

5.1 Implementación

A través de la implementación del prototipo, se logra obtener una plataforma de procesamiento de datos en tiempo real, utilizando Apache Storm como motor de procesamiento.
Las entradas de datos consideradas son la integración con sistemas externos mediante web services, como ser la aplicación de CityCop, los datos provenientes de redes sociales en este caso particular Twitter a través de la cuenta @Chorros_UY, y por último se tiene en cuenta la integración con archivos de log, que se obtienen del proyecto de procesamiento de imágenes de cámaras.
Para la ingesta de información a través de web services, se logra integrar Apache Kafka como cola de mensajes.
Se utiliza GeoKettle de la suite de Pentaho para la extracción, transformación y carga de los crímenes para su posterior análisis.
Se desarrolla una web de alertas en tiempo real, la cual muestra un mapa con los crímenes de las últimas horas, así como también se diseña e implementa un datawarehouse con el objetivo de poder visualizar los datos obtenidos, permitiendo la validación de la implementación.

Dada la experiencia de los integrantes del equipo, decidimos realizar el prototipo en la tecnología Java, y utilizar el entorno de desarrollo Eclipse.

5.1.1 Capa de Ingestión y ETL

La capa de ingestión y ETL es la encargada de tomar los datos de diversas fuentes, procesarlos y almacenarlos. En las siguientes secciones se detalla cada etapa de la misma. La definición del ETL se detalla más adelante, en la sección del datawarehouse.

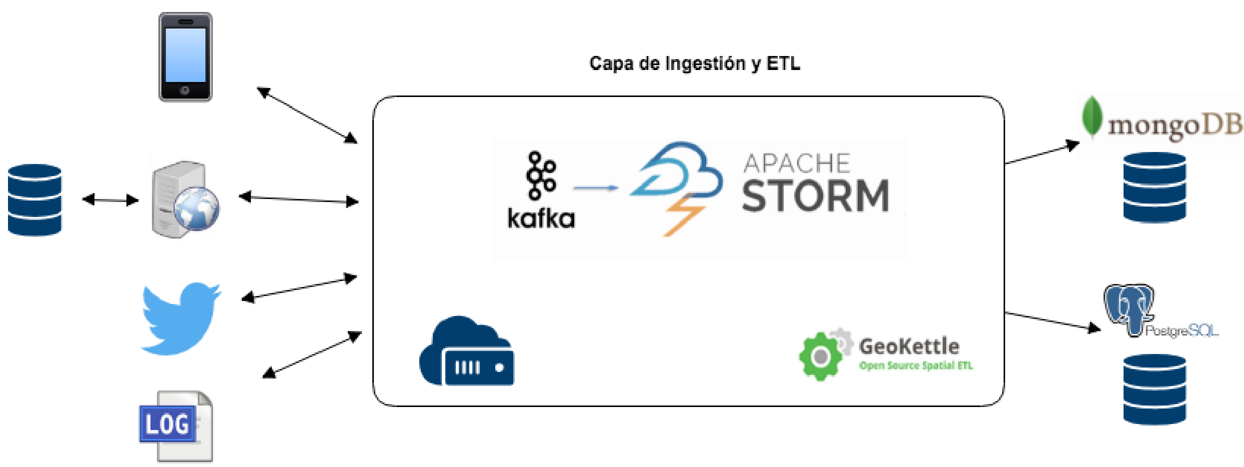


Fig. 17: Ingestión y ETL

5.1.1.1 *Entrada de datos*

Para la realización del prototipo decidimos implementar algunas de las entradas de datos definidas en la arquitectura del capítulo anterior, intentando abarcar la mayor variedad de casos.
Por lo tanto, en primer lugar consideramos implementar la integración con redes sociales, eligiendo a la red social Twitter dado que es la que tiene más facilidades y menos protección de privacidad a la hora de descargar información. Los datos de Twitter son semi estructurados, en formato JSON. Investigamos si existe alguna cuenta donde los ciudadanos reporten crímenes, encontrando la cuenta ChorrosUY, que a pesar de que no es muy utilizada sirve de prueba para nuestro prototipo. Observamos que Twitter se puede integrar con Storm fácilmente empleando la librería de Java Twitter4j.

Dado que tomamos conocimiento de la aplicación colaborativa CityCop, vemos necesario considerar como entrada de datos sistemas de información comunitarios de reportes de crímenes. Para esto implementamos una aplicación que publica un web service para que dichos sistemas lo invoquen enviando sus datos de crímenes.

La definición del web service (estructura) la hicimos teniendo en cuenta los distintos sistemas que serían posibles candidatos para proporcionar información. Esto es para estar seguros que logren integrarse con la estructura de crimen que elegimos.
Por lo tanto se analizó el Excel de la base de datos brindada por los integrantes de CityCop, en la que se tienen aproximadamente nueve mil registros de denuncias en la ciudad de Montevideo, en el período de fecha desde el 01/05/2014 al 30/04/2015 (1 año).
La estructura de la planilla es la siguiente:

Atributo	Descripción
Fecha	En formato YYYY-MM-DD HH:MM:SS
Tipo	Se detalla debajo
Descripción	Texto escrito por la persona que reporta
Latitud	Sistema de referencia EPSG:4326
Longitud	Sistema de referencia EPSG:4326
Estado	1 visible, 0 no visible
Abuso	Marcada por validación de CityCop

Dentro de los tipos de denuncias se distinguen:

- Robo a persona
- Robo a vehículo
- Robo a casa
- Robo a comercio
- Actividad sospechosa
- Homicidio
- Vandalismo
- Venta de droga
- Otros

Al investigar los datos vemos que un pilar importante es la calidad de los mismos, sobre todo los que provienen de aplicaciones del tipo VGI. Observamos que muchas de las denuncias son falsas y provienen de personas que no desean aportar datos de valor.

También accedimos a la aplicación móvil de 911, así como al sistema de denuncias en línea, para observar las acciones y datos que se tienen que dar a la hora de realizar una denuncia. Entre ellos vemos los datos del denunciante (edad, sexo), la fecha y hora de la denuncia, el tipo de víctima (transeúnte, motociclista, comercio, etc.), coordenadas, y el tipo de denuncia.
En los tipos de denuncias se encuentran: policiales (rapiña, copamiento, secuestro, herido de arma de fuego), de bomberos (incendios), tránsito, entre otros.

Finalmente llegamos a la siguiente estructura de crimen genérica, la cual es utilizada tanto en la definición del web service como en nuestro proyecto Storm (para la clase Java Crimen):

Atributo	**Descripción**
Latitude	Latitud en sistema de referencia EPSG:4326
Longitude	Longitud en sistema de referencia EPSG:4326
Place	Lugar en donde ocurre el crimen
Tipo	Tipo de crimen
Target	Victima del crimen
Date	Fecha en que ocurrió el crimen
Origin	Origen de datos del crimen
Quality	Calidad del dato
Description	Texto opcional
Source Text	Texto original

Donde Tipo de Crimen, Target, Origin y Quality están predefinidas como se muestra a continuación.

Tipo de crimen:

- Robo
- Homicidio
- Copamiento
- Rapiña
- Secuestro
- Violencia Doméstica
- Vandalismo
- No Valido
- Otros

Origin: se consideran las distintas fuentes de datos

- Twitter
- 911
- CityCop
- Sistema denuncias en línea

Quality: determinamos un criterio de calidad el cual considera de donde proviene el dato y si hay coincidencias con otros crímenes recientemente ingresados al sistema.
Se define como calidad Alta, Media, Baja.

En Java se definen como enumerados que pueden ser extensibles, por lo que se pueden agregar o quitar.

5.1.1.2 Topología Storm

En esta sección mostramos el diseño de la topología Storm, la cual es un grafo que contiene spouts y bolts, y representa nuestro motor de procesamiento. En la figura 18 se puede contemplar dicho diagrama.

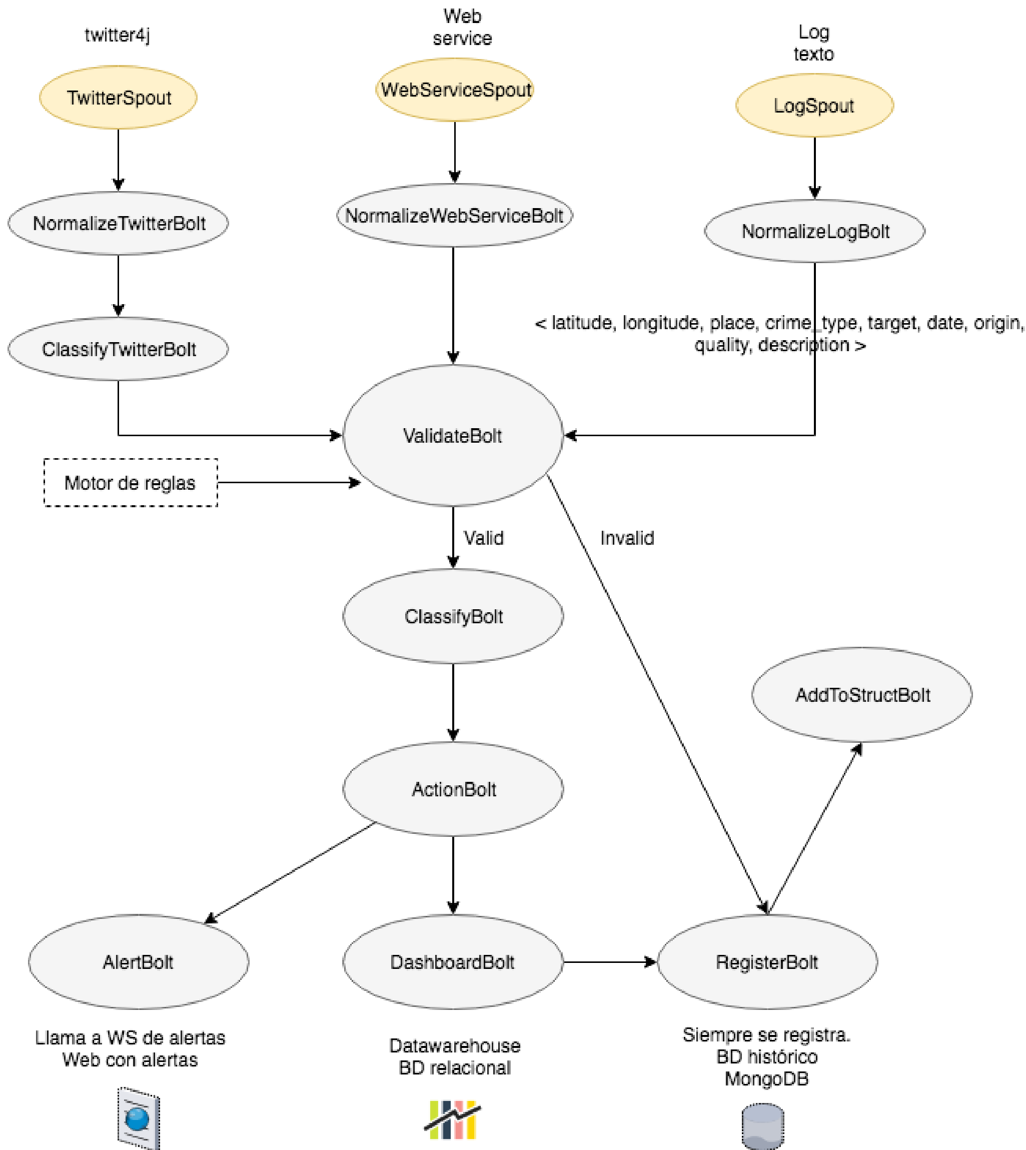


Fig. 18: Topología Storm

En este diagrama se definen en primer lugar los spouts, que representan las diversas entradas de datos al sistema. Esta topología es escalable y extensible, dado que si se desea agregar una nueva entrada de datos se puede concebir fácilmente agregando un nuevo nodo spout.
En el diseño de este prototipo se tienen un spout "TwitterSpout" para la integración con Twitter, un spout para la integración con sistemas externos, donde esta última se llevará a cabo a través de web services, y un spout para la integración con sistemas que generen archivos de log. Se definirán más específicamente en las siguientes secciones.

Luego los bolts son los encargados del procesamiento. Se determina un bolt asociado a cada spout donde se normalizarán los datos. Es decir, dicho paso generará la tupla que seguirá recorriendo la topología. La estructura de esta tupla es el crimen definido anteriormente en la tabla de la sección

anterior.

Posteriormente a la normalización de los datos, se continua con el spout "ValidateBolt" el cual se encarga de validar la tupla, según un motor de reglas predefinido. En nuestro caso, nuestro motor de reglas descarta los tweets que no tengan ciertas palabras relacionadas a un crimen en su texto. Esto se hace comparando con un diccionario definido previamente.
A su vez damos la posibilidad de descartar fuentes de datos, es decir, si no se quieren tener en cuenta los datos proveniente de Twitter, se modifica en el archivo de configuración cierto parámetro para no tenerlo en cuenta y marcar al crimen como inválido.
Este motor es extensible, por lo que se podrán agregar las reglas que se deseen en un futuro. En el caso que la tupla sea inválida se guarda en el histórico y se finaliza el procesamiento.

Si la tupla se considera válida se prosigue al siguiente bolt "ClassifyBolt", el cual se ocupa de asignarle la calidad/confiabilidad al dato. En este contexto consideramos que un dato proveniente de Twitter y CityCop tienen calidad baja, mientras que un dato de los sistemas 911 y similares tienen prioridad media. La calidad Alta es reservada para los crímenes con coincidencia.
Asimismo se encarga del cruzamiento de datos, buscando similitudes entre crímenes. Si dicha comparación es exitosa, aumenta la calidad del dato, y se asocian los crímenes. Se comenta con más detalle en la sección "Cruzamiento de datos".

El procesamiento continúa con el spout "ActionBolt", el cual emite la tupla al bolt correspondiente de salida. Entre los bolts de salida se encuentran el "AlertBolt" encargado de generar la alerta al sistema externo, "DashboardBolt" que persistirá el crimen en la base de datos PostgreSQL y "RegisterBolt" que guardará el crimen en la base de datos no relacional de histórico MongoDB.
Específicamente, el bolt "AlertBolt" consumirá un web service SOAP enviando la información para que ésta se muestre en un mapa contenido en una web de alertas en tiempo real. Se detallará en la sección "Web de alertas".

El bolt "AddToStructBolt" se definirá en la sección "Cruzamiento de datos".

La topología de Storm se implementa utilizando el lenguaje de programación Java, y utilizando la dependencia Apache Storm.
Para crear un Spout en Java hay que definir una clase que implemente la interfaz IRichSpout, y a su vez implementar los métodos de la misma que son: open, close, activate, deactivate, nextTuple, ack, fail.

- open: se invoca cuando se inicializa una task para el bolt dentro del worker en el cluster. A este método se le pasan tres parámetros:
  - conf: la configuración de Storm para este spout. Es la configuración proporcionada por la topología combinada con la configuración del cluster.
  - context: este objeto se puede utilizar para obtener información del lugar de esta task dentro de la topología, así como también obtener los identificadores de la task e información de entrada y salida.
  - collector: se usa para emitir tuplas en el spout.
- close : se invoca cuando se va a "apagar" el spout.
- activate: se invoca cuando se quiere activar un spout que estaba desactivado.
- deactivate: se invoca cuando se desactiva un Spout, el método nextTuple no puede ser invocado mientras el componente este desactivado.
- nextTuple: este método se llama cuando se le solicita al spout que emita una tupla del "collector de salida" outputCollector. Es decir, cuando se avisa al spout que pueden haber tuplas para procesar.
- ack: Storm determinó que la tupla con el id de mensaje que se le pasa a esta método ha sido procesada por completo.

- fail: la tupla emitida por el spout con id de mensaje X, no fue procesada por completo. Esto requeriría que la tupla sea retransmitida por el componente.
- declareOutputFields: declara el esquema de la salida para todos los streams de el spout actual.

Asimismo, para realizar un Bolt hay que implementar la interfaz IRichBolt o IBasicBolt y los siguientes métodos: prepare, execute, cleanup, declareOutputFields.

- prepare: Es análoga al open en los spouts; se llama cuando se inicializa la task en la que se encuentra el componente. Los parámetros con los que se invoca también son análogos:
  - stormConf: es la configuración de la topología combinada con la configuración del cluster proporcionada por Storm para el componente.
  - context: este objeto tiene información sobre la ubicación de la task que contiene el componente dentro de la topología.
- execute: se encarga de procesar una tupla de entrada. El objeto tuple que se le pasa como parámetro tiene metadata con información de la procedencia de la tupla (componente/stream/task de donde proviene). El valor de la tupla se obtiene haciendo tuple.getValue(). Puede ser emitida utilizando el outputCollector proporcionado por el método prepare. En el mismo momento es necesario que la tupla sea acked or failed.
- cleanUp: se utiliza cuando se va a apagar el spout. No se garantiza su invocación dado que el supervisor puede matar los workers previamente.
- declareOutputFields: declara el esquema de salida para los streams de este componente.

En nuestro proyecto Topology.java es la clase principal la cual conecta los spouts y bolts. En el siguiente código se puede visualizar la implementación de dicha clase (no se incluye el total del código sino lo relevante para el entendimiento de la clase).

```java
// Topology.java
import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.kafka.KafkaSpout;
import org.apache.storm.topology.TopologyBuilder;

public class Topology {

   public static void main(String[] args) throws Exception {
      final TopologyBuilder builder = new TopologyBuilder();

      //Kafka input
      SpoutBuilder spoutBuilder = new SpoutBuilder(configs);
      KafkaSpout kafkaSpout = spoutBuilder.buildKafkaSpout();

      builder.setSpout("kafkaSpout", kafkaSpout,
          Integer.parseInt(configs.getProperty(Keys.KAFKA_SPOUT_PARALL)));
      builder.setBolt("normalizeDenunciasBolt", new NormalizeDenunciasBolt(),
          Integer.parseInt(configs.getProperty(Keys.NORMALIZE_DENUNCIAS_PARALL)))
        .shuffleGrouping("kafkaSpout");

      //Twitter input
      builder.setSpout("twitterSpout", new TweetsStreamingConsumerSpout(),
          Integer.parseInt(configs.getProperty(Keys.TWITTER_SPOUT_PARALL)));
      builder.setBolt("normalizeTwitterBolt", new NormalizeTwitterBolt(),
          Integer.parseInt(configs.getProperty(Keys.NORMALIZE_TWITTER_PARALL)))
        .shuffleGrouping("twitterSpout");
```

```java
    builder.setBolt("classifyTwitterBolt", new ClassifyTwitterBolt(crimesDictionary),
        Integer.parseInt(configs.getProperty(Keys.CLASIFY_TWITTER_PARALL)))
      .shuffleGrouping("normalizeTwitterBolt");

    //Validate
    builder.setBolt("validateBolt", new ValidateBolt(InvalidOrigins()),
        Integer.parseInt(configs.getProperty(Keys.VALIDATE_PARALL)))
      .shuffleGrouping("classifyTwitterBolt").shuffleGrouping("normalizeDenunciasBolt");

    //Classify
    builder.setBolt("classifyBolt", new ClassifyBolt(),
        Integer.parseInt(configs.getProperty(Keys.CLASIFY_PARALL)))
      .shuffleGrouping("validateBolt", "streamClassify");

    //Action
    builder.setBolt("actionBolt", new ActionBolt(),
        Integer.parseInt(configs.getProperty(Keys.ACTION_PARALL))).shuffleGrouping("classifyBolt");

    //Alert - Dashboard - Register
    builder.setBolt("alertBolt", new AlertBolt() ,
        Integer.parseInt(configs.getProperty(Keys.ALERT_PARALL))).shuffleGrouping("actionBolt",
        "streamAlert");
    builder.setBolt("dashboardBolt", new DashboardBolt(),
        Integer.parseInt(configs.getProperty(Keys.DASHBOARD_PARALL)))
      .shuffleGrouping("actionBolt", "streamDashboard");
    builder.setBolt("registerBolt", new RegisterBolt(),
        Integer.parseInt(configs.getProperty(Keys.REGISTER_PARALL)))
      .shuffleGrouping("validateBolt",
         "streamRegister").shuffleGrouping("dashboardBolt");

    //Add to structure
    builder.setBolt("addToStructBolt", new AddToStruct(),
        Integer.parseInt(configs.getProperty(Keys.ADD_STRUCT_PARALL))).shuffleGrouping("registerBolt");

  }
}
```

La clase Topology.java contiene el programa principal el cual es el punto de entrada a la topología.
El método TopologyBuilder es el encargado de construir la topología.
Se determinan los nodos de la misma utilizando los métodos setSpout y setBolt, descriptos previamente. Éstos toman como entrada un id definido por el usuario, un objeto que contiene la lógica de procesamiento, y la cantidad de paralelismo que se quiere para dicho nodo, el cual es opcional. El parámetro de paralelismo, refiere a cuantos hilos debe ejecutar ese nodo en el cluster, y en caso de que se omita Storm solo asignará un hilo para ese nodo.
Por ejemplo el spout que se define para Twitter, tiene como id "twitterSpout", la clase que lo implementa es "TweetsStreamingConsumerSpout()", y el nivel de paralelismo se obtiene del parámetro de configuración "TWITTER_SPOUT_PARALL", que se detalla en el anexo de configuración.

El método setBolt retorna un objeto del tipo InputDeclarer que se usa para definir las entradas al bolt. Por ejemplo en el caso del bolt de normalización de Twitter, el componente "normalizeTwitterBolt" declara que quiere leer todas las tuplas emitidas por "twitterSpout" utilizando shuffleGrouping.

Storm provee dos modos de operación: modo local y modo distribuido. En el modo local, Storm simula con hilos los distintos nodos del cluster, el cual es útil en la etapa de pruebas y desarrollo de topologías ya que permite probar el código y ajustar los parámetros de configuración.
En modo distribuido, Storm funciona como un cluster de máquinas. Cuando envía una topología al maestro, también envía todo el código necesario para que sea ejecutada. El maestro se encargará de distribuir su código y asignar workers para ejecutar su topología. Si los workers caen, el maestro los reasignará a otro lugar.

A continuación se muestra como se realiza el submit de la topología en modo local.

```
final Config conf = new Config();
conf.setDebug(true);
conf.registerSerialization(Crime.class);
conf.setNumWorkers(Integer.parseInt(configs.getProperty(Keys.NUM_OF_WORKERS)));

final LocalCluster cluster = new LocalCluster();
cluster.submitTopology("crimesTopology", conf, builder.createTopology());
```

Se crea el cluster en proceso a través del objeto LocalCluster. El submit de la topología se realiza llamando al método submitTopology, el cual toma como argumentos el nombre de la topología, en este caso "crimesTopology", la configuración correspondiente como así también la topología.
La configuración incluye el número de workers, es decir cuántos procesos se quieren asignar en el cluster para ejecutar la topología. Y la opción de debug que cuando se setea en True hace que Storm escriba en el log todos los mensajes emitidos por un componente, lo cual puede ser utilizado en el proceso de desarrollo.

El modo distribuido es similar al modo local, pero se agrega la configuración del parámetro setMaxSpoutPending, el cual indica la cantidad máxima de tuplas que pueden estar ejecutando al mismo tiempo dentro de Storm. A su vez, se utiliza StormSubmitter para realizar el submit correspondiente de la topología.
Luego se debe crear un jar que contenga el código y todas las dependencias, el cual se utilizará para hacer el submit de la topología usando el cliente storm. [35]

```
Config conf = new Config();
conf.setDebug(false);
conf.setNumWorkers(Integer.parseInt(configs.getProperty(Keys.NUM_OF_WORKERS)));
conf.setMaxSpoutPending(5000);

StormSubmitter.submitTopology("crimesTopology", conf, builder.createTopology());
```

Cabe destacar que una topología ejecuta indefinidamente hasta que el usuario la termine.

5.1.1.3 Integración con Twitter

El sistema que desarrollamos considera como una de sus fuentes los datos provenientes de Twitter, en el contexto de reportes de crímenes desde determinadas cuentas.

Twitter es un servicio de redes sociales que proporciona una plataforma para enviar y recibir tweets de usuarios. Los usuarios registrados pueden leer y publicar tweets, pero los usuarios no registrados solo pueden leerlos. El Hashtag se utiliza para categorizarlos por palabra clave al agregar el símbolo # antes de la palabra relevante.

Twitter provee la API "Twitter Streaming API", una herramienta basada en un servicio web para recuperar los tweets enviados por las personas en tiempo real. Se puede acceder a Twitter Streaming API en cualquier lenguaje de programación.
twitter4j [36] es una librería no oficial open source Java, que proporciona un módulo basado en Java para acceder fácilmente a la API de Twitter Streaming. Para acceder a esta API, se debe iniciar sesión en la cuenta de desarrollador de Twitter y obtener los siguientes detalles de autenticación de OAuth: Customerkey, CustomerSecret, AccessToken y AccessTookenSecret.

Para poder obtener los tweets que referencian a un usuario de Twitter se necesita comunicarle a la API de Twitter el Id de dicho usuario, el cual se puede obtener desde distintas páginas web como por ejemplo www.gettwitterid.com.
En nuestro caso encontramos una cuenta de Twitter "ChorrosUy" en donde los usuarios realizan publicaciones de distintos crímenes, y a su vez el propietario de la cuenta también hace denuncias a través de ese medio.

La información descargada de Twitter que nos interesa, para cada tweet, es la siguiente:

- Identificador único
- Texto.
- Nombre de usuario que lo ha publicado.
- La posición, si tiene, en forma de latitud y longitud.
- La fecha y la hora de ubicación.

Para el procesamiento del contenido del tweet, consideramos en primer lugar integrar alguna API o herramienta de procesamiento de lenguaje natural. La estrategia a utilizar era la siguiente: separar las palabras contenidas en el texto del tweet y para cada una obtener el lema. El lema en la teoría lexicográfica es la totalidad de formas que toma una palabra y que se toma como representante de todas las variaciones morfológicas de la palabra. Es decir, el lema de una palabra es la palabra que aparece en el diccionario como entrada para una definición.
A partir del lema de todas las palabras se buscan algunos lemas concretos que pudieran caracterizar el tweet (por ejemplo robar, asaltar, rapiñar) y a partir de eso se define a qué tipo de crimen hace referencia, o en el caso que no se pueda queda como indeterminado (no definido).
La ventaja que se obtiene al utilizar esta estrategia de lemanización es que no se tiene que tener un diccionario con todas las palabras que representan crímenes, además de las distintas formas que puede llegar a tener una palabra. Por ejemplo si el tweet hace mención de una rapiña, en el texto puede aparecer la palabra de distintas maneras: rapiña, rapiñaron, rapiñaban. Lo que no varía en ninguna es el lema rapiñar.

Para la estrategia de lemanización del documento investigamos dos APIs de procesamiento de lenguaje natural, Apache OpenNLP y Stanford CoreNLP. Para la herramienta Stanford CoreNLP al momento que la investigamos no tenia soportado la lemanización para el idioma español, por eso fue descartada. Por otra parte intentamos integrar la otra API a nuestro proyecto de Java y tuvimos algunas dificultades. Por lo tanto al ver que nos iba a llevar más tiempo de lo pensado utilizar una herramienta de procesamiento de lenguaje natural en nuestra solución, descartamos esta posibilidad.

Finalmente resolvimos implementar un diccionario con las palabras claves que vamos a buscar dentro del contenido de un tweet para clasificarlo. Para cada palabra clave también guardamos las distintas formas en las que puede aparecer, así como también sinónimos.
Cuando ingresa un tweet al bolt ClassifyTwitterBolt se divide en palabras y se busca si coincide con alguna palabra del diccionario. Dado que el tweet esta diseñado para ser corto (140 caracteres máximo) esto no es un procesamiento demasiado costoso.
Para realizar este pre proceso sobre el texto se ha implementado una clase diccionario que lee un archivo (llamado diccionario.xml) que contiene las listas de palabras a buscar, el cual es extensible. La estructura del diccionario se puede apreciar en la figura 19

```
<?xml version="1.0" encoding="UTF-8"?>
<crimeTypes>
    <crimeType name="robo">
        <synonym>robo</synonym>
        <synonym>robaron</synonym>
        <synonym>hurto</synonym>
        <synonym>asalto</synonym>
        <synonym>arrebato</synonym>
        <synonym>abigeato</synonym>
        <synonym>asaltado</synonym>
        <synonym>asaltó</synonym>
    </crimeType>
    <crimeType name="rapiña">
        <synonym>rapiña</synonym>
        <synonym>rapiñaron</synonym>
        <synonym>rapiñaban</synonym>
    </crimeType>
    <crimeType name="homicidio">
        <synonym>homicidio</synonym>
    </crimeType>
    <crimeType name="secuestro">
        <synonym>secuestro</synonym>
    </crimeType>
</crimeTypes>
```

Fig. 19: Estructura diccionario

Se utilizó una archivo XML para facilitar la lectura y a su vez para establecer sencillamente jerarquía entre las palabras, teniendo así las palabras principales y los sinónimos.

5.1.1.4 Integración con Kafka

Este plataforma debe contemplar la comunicación con sistemas o aplicaciones externas para la ingesta de datos. Para esto se implementa una aplicación que expone un web service con la idea que sea consumido por estos sistemas. La estructura de dicho servicio se detalla en la sección anterior "Entrada de datos", por lo que se envía un crimen por invocación.
Estos crímenes recibidos se deben enviar a la topología Storm para su procesamiento, siendo en este caso Apache Kafka el intermediario.

Storm permite la integración con colas de mensajes, proporcionando una serie de spouts preconstruidos para leer datos de colas externas.
¿Por qué elegimos Kafka y no otra cola de mensajes? Por diseño, Kafka es más adecuado para escalar que los sistemas de colas tradicionales debido a que maneja cada partición de un topic como un registro (log, un conjunto ordenado de mensajes). Puede dividir entre los consumidores por partición y enviar los mensajes en lotes. Kafka maneja la paralelización de consumidores mejor que las colas tradicionales, e incluso puede manejar las fallas para un consumidor en un grupo de

consumidores. [37]
Ante una suma importante de mensajes almacenados en disco no se ve afectado el rendimiento, tanto al publicar como al subscribir.

En primera instancia, se crea el topic de Kafka "incoming" donde serán publicados los mensajes. Esto se lleva a cabo desde la consola ejecutando los comandos correspondientes. Se prueba intercambiar mensajes desde consola, iniciando el broker de Kafka y los procesos productor y consumidor, verificando así que la instalación y configuración es correcta.
En el manual de usuario se detallan los comandos para hacerlo.

Del lado del proyecto principal donde se desarrolla la topología Storm, usamos la librería Apache Kafka en la versión 0.8.2.1.
Para esto creamos un Kafka Spout, una implementación de un spout regular que lee la información del cluster Kafka, es decir que lee del topic que creamos anteriormente ("incoming"). Para conectarse con el cluster se debe proporcionar la siguiente información: nombre del topic del cual se obtendrán los mensajes, la ruta donde se encuentra ZooKeeper y el grupo de consumidores. A continuación se puede visualizar la creación de un Spout de Kafka. [38]

```
public KafkaSpout buildKafkaSpout() {

   BrokerHosts hosts = new ZkHosts(configs.getProperty(Keys.KAFKA_ZOOKEEPER));
   String topic = configs.getProperty(Keys.KAFKA_TOPIC);
   String zkRoot = configs.getProperty(Keys.KAFKA_ZKROOT);
   String groupId = configs.getProperty(Keys.KAFKA_CONSUMERGROUP);

   SpoutConfig spoutConfig = new SpoutConfig(hosts, topic, zkRoot, groupId);
   spoutConfig.scheme = new SchemeAsMultiScheme(new StringScheme());
   KafkaSpout kafkaSpout = new KafkaSpout(spoutConfig);
   return kafkaSpout;
}
```

Por otro lado se cuenta con el proyecto "crime-integration", encargado de publicar un web service y ante la invocación del mismo, enviar los datos a Kafka. Para ello se utilizó la librería kafka-clients en su versión 0.8.2.1.
A continuación se puede contemplar la publicación en Kafka, cuando se recibe un nuevo crimen:

```
Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"localhost:9092");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
   "org.apache.kafka.common.serialization.StringSerializer");
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
   "org.apache.kafka.common.serialization.StringSerializer");

props.put("retries", 0);
props.put("batch.size", 16384);
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432);

Thread.currentThread().setContextClassLoader(null);

final Producer<String, String> producer = new KafkaProducer<>(props);

Gson g = new Gson();
String jsonString = g.toJson(crime);

producer.send(new ProducerRecord<String, String>("incoming", "0", jsonString)).get();
producer.close();
```

Se especifica la configuración correspondiente:

- BOOTSTRAP_SERVERS_CONFIG: indica la dirección del broker Kafka. En el caso de que esté ejecutando en un cluster, se ponen las direcciones separadas por una coma (,), por ejemplo localhost:9091,localhost:9092.
- KEY_SERIALIZER_CLASS_CONFIG: indica la clase que será utilizada para serealizar el objeto clave.
- VALUE_SERIALIZER_CLASS_CONFIG: indica la clase que será utilizada para serealizar el objeto valor. En nuestro caso el valor es un String por lo que usamos la clase StringSerializer.

El crimen se convierte al formato JSON y se envía utilizando la clase KafkaProducer. [39]

Durante el desarrollo de la integración, surgieron algunas dificultades. En primer lugar, la documentación no dejaba claro la configuración necesaria para crear el productor. Luego de resuelta dicha dificultad, logramos publicar en Kafka, pero dándonos un error conocido vinculado a las versiones, el cual logramos resolver probando con las distintas versiones de Kafka y kafka-clients.

5.1.1.5 Cruzamiento de datos

En esta etapa se verifica si existen coincidencias entre crímenes que son reportados por distintos medios.
Este cruzamiento de datos en tiempo real se lleva a cabo en el bolt ClassifyBolt, utilizando un Manager de crímenes que proporciona operaciones para poder identificar si un crimen nuevo se corresponde con alguno que ya fue procesado. Dicho Manager es una clase Singleton la cual contiene una estructura que guarda los crímenes más recientes y los mantiene durante un tiempo x configurable. Cuenta con operaciones para obtener los crímenes del mismo tipo, hora y ubicación geográfica.

Como Storm está pensado para que haya varias instancias de un mismo bolt cuando la topología está en ejecución, decidimos utilizar una estructura para guardar los crímenes que se pueda acceder de manera concurrente, y a su vez facilite la búsqueda por los atributos que necesitemos. Para esto evaluamos la posibilidad de dos tipos de estructuras: ConcurrentHashMap y SynchronizedHashMap.

- ConcurrentHashMap: este tipo de estructura se recomienda utilizar cuando se tiene alto nivel de concurrencia. Las lecturas se realizan de forma rápida mientras que las escrituras (inserciones) se realizan con un bloqueo. No hay bloque a nivel de objeto, es decir que se puede trabajar sobre los elementos de la estructura sin estar bloqueándola. Esta estructura lanza una excepción si un hilo intenta modificar mientras otro está iterando sobre él.
- SynchronizedHashMap: en este caso la sincronización es a nivel de objeto. las operaciones de lectura y escritura necesitan bloquear toda la estructura, esto implica una sobrecarga de rendimiento. El funcionamiento este implica que solo un hilo pueda estar utilizando la estructura a la vez.

Dado el alto nivel de concurrencia y la necesidad de utilización de la estructura decidimos implementar ConcurrentHashMap. Dicha estructura es de tipo <clave, valor>, en la cual la clave es el tipo de crimen y el valor una lista de todos los crímenes de ese tipo, la lista es una ConcurrentLinkedQueue. De esta manera obtenemos que la búsqueda de crímenes por tipo de crimen se haga en orden 1.

Para poder asociar los crímenes entrantes con los que ya están procesados, primero se filtra en la estructura todos los crímenes del mismo tipo del crimen que se está procesando; esto se hace primero porque es la operación menos costosa y filtra una gran cantidad de datos.
Luego a partir de una lista con todos los crímenes del mismo tipo se filtra por los que tengan una fecha y hora aproximada a la del nuevo crimen (la diferencia de hora se puede configurar a la que se considere adecuada).

Por último, a partir de una lista resultado de los filtros anteriores, se buscan los que tengan proximidad geográfica (se detalla a continuación).

Al encontrar coincidencia de tipo, fecha y hora, y proximidad geográfica, se asocian dichos crímenes a nivel de base de datos, tanto en PostgreSQL como MongoDB, como forma de unificarlos. A su vez, la veracidad de los crímenes aumentan, dicho de otra forma, la calidad de ese dato aumenta.

Por último, para mantener actualizada la estructura de crímenes implementamos un Timer que ejecuta cada determinado tiempo configurable, encargado de eliminar los crímenes que ya expiraron. La cantidad de tiempo que tiene que transcurrir para que un crimen se considere expirado es configurable según las necesidades.

Filtro distancia

Para realizar el filtro de proximidad geográfica utilizamos JTS Topology Suite (JTS) [40]. Es una librería de código abierto Java, que proporciona un modelo de objetos para la geometría planar junto con un conjunto de funciones geométricas fundamentales. JTS cumple con la especificación de Simple Features Specification para SQL publicada por el Open GIS Consortium. JTS está diseñado para ser utilizado como un componente central del software geomático basado en vectores, como los sistemas de información geográfica. También se puede usar como una librería de propósito general que proporciona algoritmos en geometría computacional.
Es ampliamente utilizada en el software SIG de código libre con funciones de análisis espacial, consultas avanzadas y creación de topología.

También fue necesario el uso de GeoTools [41], otra librería open source Java, que proporciona métodos que cumplen con los estándares para la manipulación de datos geoespaciales, por ejemplo para implementar Sistemas de Información Geográfica. Dentro de sus core features se encuentra JTS para el soporte del objeto Geometry.

Por lo tanto, luego del filtrado por tipo de crimen y fecha y hora, se continua con el filtrado por ubicación. Es decir, dadas las coordenadas del nuevo crimen ingresado a la topología, se verifica si el mismo "está cerca" de algún crimen que ya se encuentra en la estructura del sistema. Para esto se utilizan las librerías definidas anteriormente.
En primer lugar se crea el punto del crimen con las coordenadas pertinentes, y luego se compara dicho punto utilizando el método JTS.orthodromicDistance, el cual nos devuelve la distancia que posteriormente transformamos a metros. La distancia en metros que consideramos que se tiene que dar para que dos puntos se consideren cercanos se puede configurar desde el archivo de configuración. Por defecto asignamos el valor de 100 metros.

5.1.2 Capas de Análisis y Visualización

En las siguientes secciones definimos los modelos de bases de datos, así como también el diseño del datawarehouse y la web de alertas.
Cabe destacar que la implementación del prototipo está centrada en la capa de ingestión y ETL, y que el desarrollo de las capas de análisis y visualización se realizan para la validación del mismo. Es por este motivo que se utilizaron herramientas conocidas por lo integrantes, facilitando la implementación.

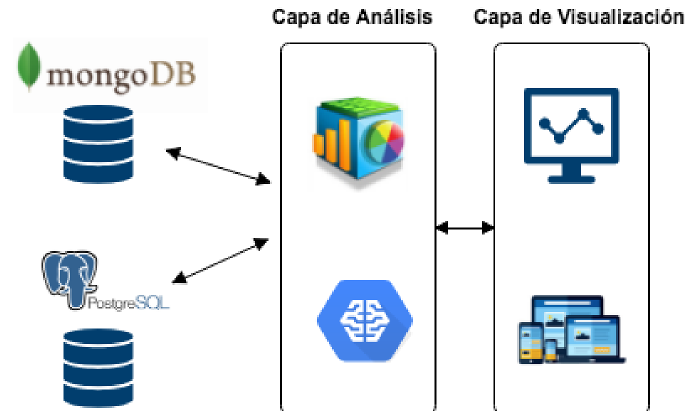


Fig. 20: Análisis y visualización

5.1.2.1 Modelos de datos

Contamos con dos bases de datos de las cuales una es relacional y la otra no relacional.
A continuación presentamos el modelo de datos de cada una de ellas.

Base de datos relacional

En la figura 21 se presenta el diseño de la base de datos PostgreSQL, que consta de la entidad principal "Crime" que identifica a los crímenes, y las entidades "CrimeType" que contiene los tipos de crímenes, "Quality" para el atributo de calidad y "Origin" para el origen de los datos. Esta base de datos se utiliza en la etapa de ETL para generar la base de datos del datawarehouse.

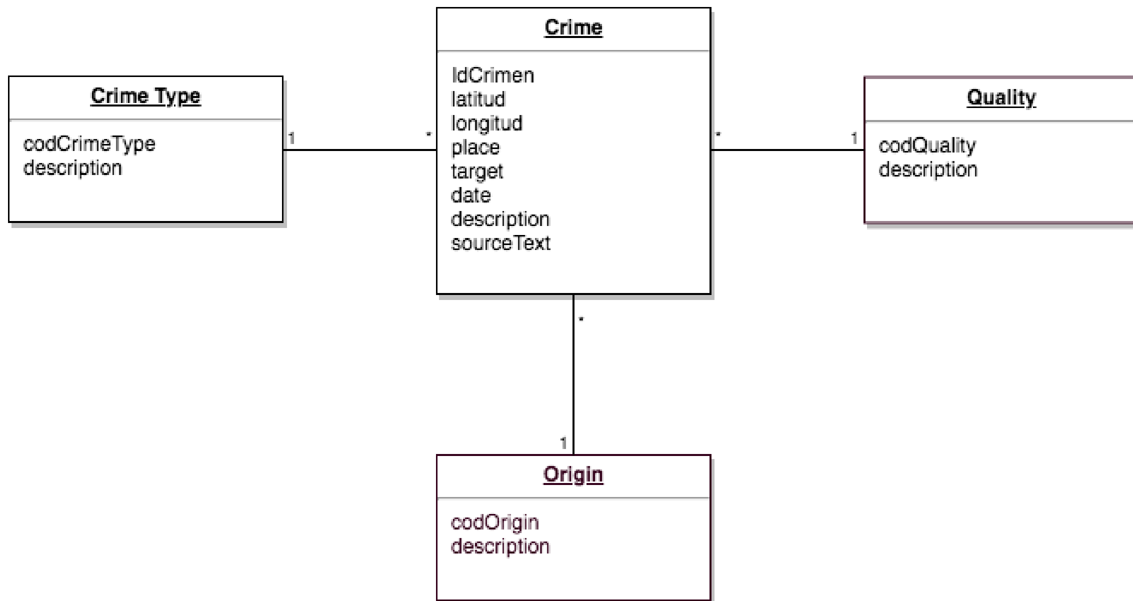
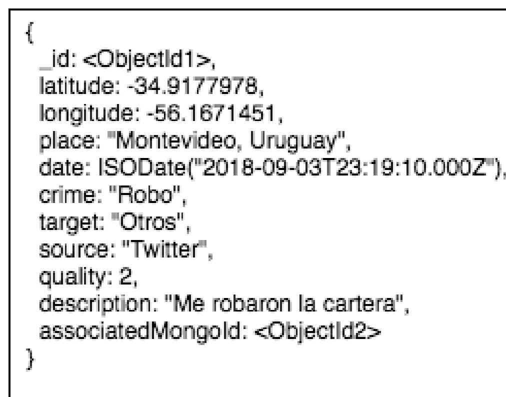


Fig. 21: Diseño base de datos PostgreSQL

Base de datos no relacional

En la figura 22 se presenta la colección "crimes" de la base de datos MongoDB, la cual se utiliza como histórico de datos.



```
{
  _id: <ObjectId1>,
  latitude: -34.9177978,
  longitude: -56.1671451,
  place: "Montevideo, Uruguay",
  date: ISODate("2018-09-03T23:19:10.000Z"),
  crime: "Robo",
  target: "Otros",
  source: "Twitter",
  quality: 2,
  description: "Me robaron la cartera",
  associatedMongoId: <ObjectId2>
}
```

Fig. 22: Diseño documento crime de MongoDB

5.1.2.2 Datawarehouse

Para tener noción de qué información le resultaría interesante conocer al negocio sobre las denuncias de crímenes, analizamos informes del Observatorio Nacional sobre Violencia y Criminalidad del Uruguay [22], pertenecientes al Ministerio del Interior.
A partir de esto, conformamos el siguiente requerimiento:

Requerimientos funcionales

Un requerimiento válido sería el siguiente:

Se quiere analizar la información de crímenes realizados en el Uruguay a través del tiempo.
Interesa visualizar la cantidad de crímenes, la variación de los mismos y el promedio.

Los indicadores antes mencionados se quieren visualizar según el tipo de crimen, tipo de víctima, el sexo y rango de edad de la víctima, ubicación geográfica y momento en que se cometió.

Los tipos de crímenes son robo, rapiña, homicidio y secuestro. A su vez interesa ver para cada tipo de crimen los distintos subtipos, por ejemplo un hurto puede ser a una casa de familia, vehículo, documentos, etc.

Los tipos de víctima son transeúnte, comercio, motociclista, repartidor, taximetrista, automovilista, ómnibus, casa de familia, menor, otros.

Sobre la ubicación geográfica interesan los niveles de país, departamentos, localidades y barrios.

El momento en el tiempo en el cual es realizado el crimen se quiere visualizar a nivel de hora. También debe existir la posibilidad de ver la información agrupada por día de la semana, mes, trimestre y año o por rangos de horarios o meses. Los rangos de horarios son de 06:00 a 14:00 hs, de 14:00 a 22:00 hs y de 22:00 a 06:00 hs.

Se quiere visualizar la información en mapas, dashboards.

Interesa también en este caso observar la información según la fuente de dato del que proviene.

Dado que estamos realizando un prototipo decidimos acotar el requerimiento a implementar al siguiente:
"Cantidad de crímenes por tipo, fecha y barrio, según la fuente de datos. "

Modelo conceptual

Las dimensiones identificadas son:

- Tipo crimen
- Tiempo
- Ubicación geográfica
- Origen

La estructura de cada dimensión se puede apreciar en la figura 23.

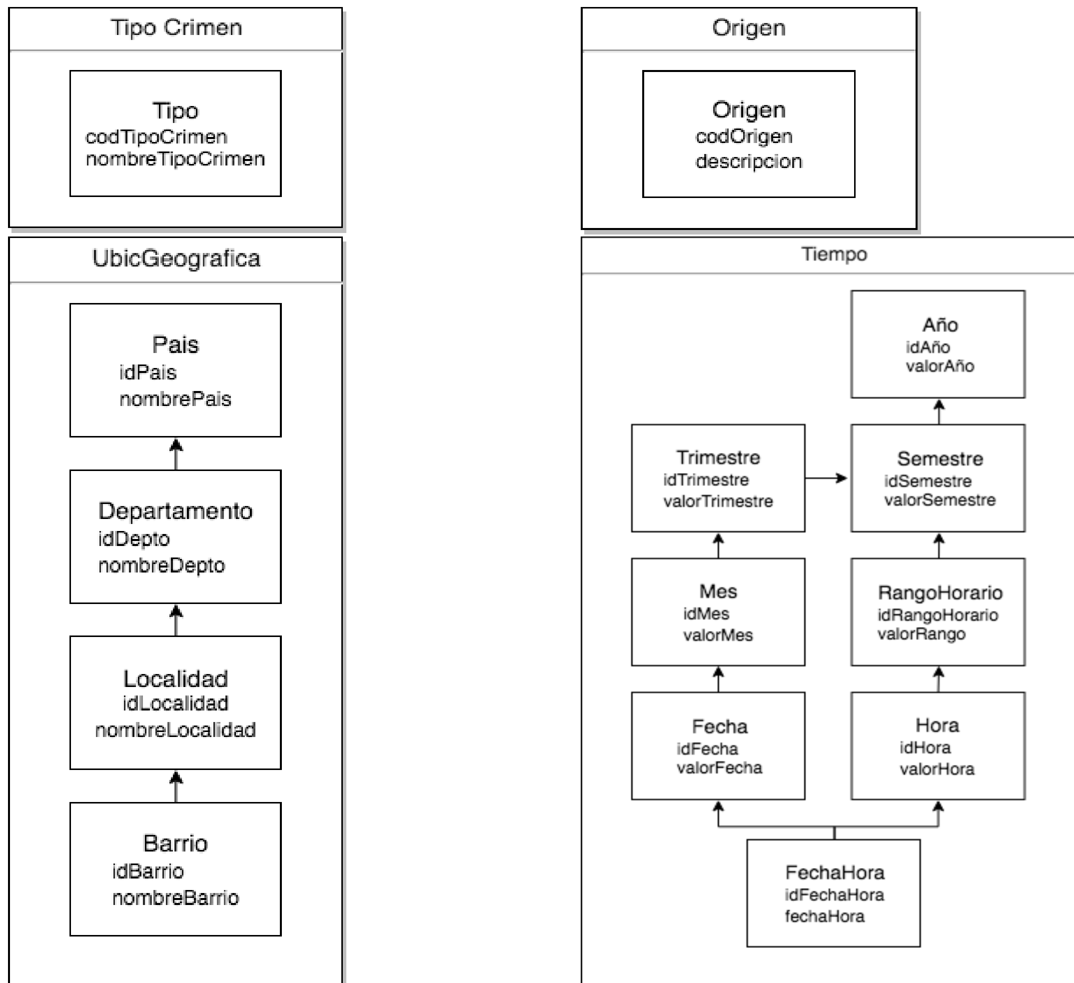


Fig. 23: Dimensiones

Relaciones dimensionales

La medida a considerar será cantidad de crímenes, los cuales se quieren visualizar según tipo, tiempo, ubicación geográfica y origen.
El diagrama de relaciones dimensionales se puede visualizar en la figura 24.

En este caso no es necesario hacer un Roll-up de las medidas.

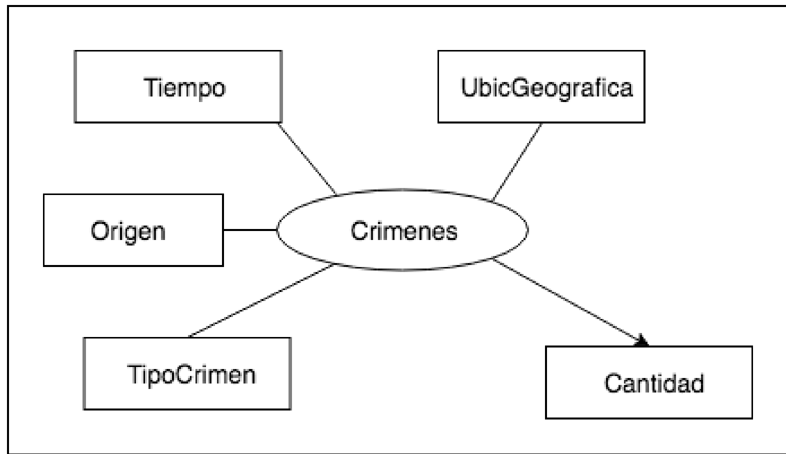


Fig. 24: Relaciones dimensionales

Diseño lógico

Se define el modelo estrella para la relación dimensional definida anteriormente, en la figura 25.

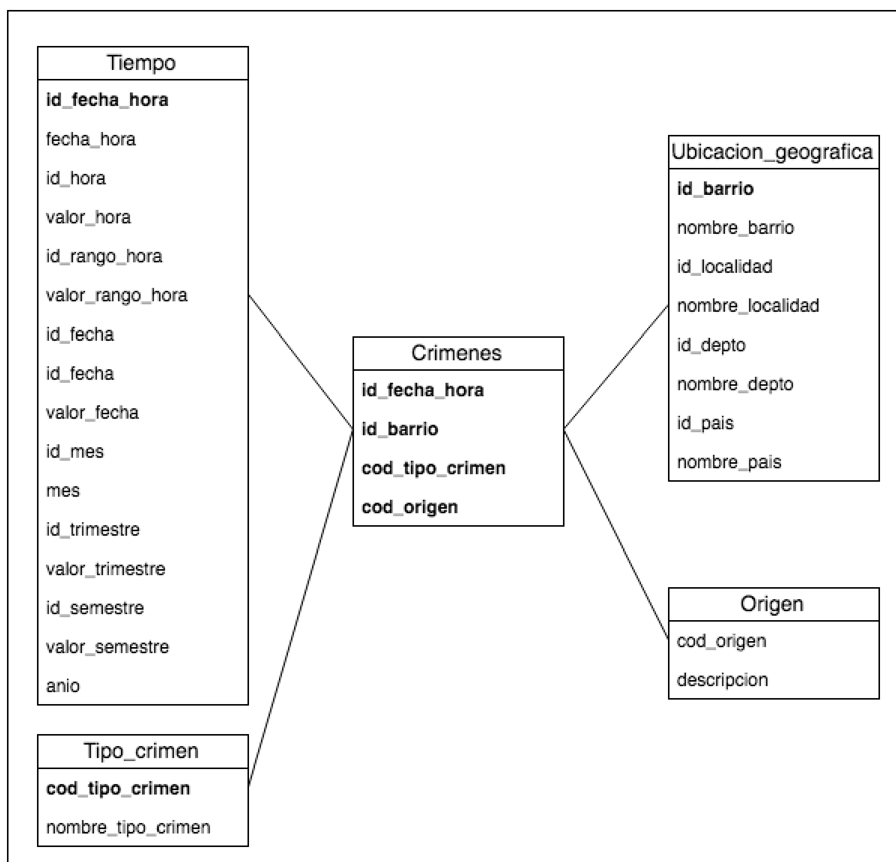


Fig. 25: Diseño lógico

Implementación

Al considerar las herramientas para realizar el datawarehouse, investigamos las distintas posibilidades basándonos en el cuadrante mágico de Gartner [42] y las que sean open source o con versión community. Observamos que la mayoría son herramientas pagas, o con trial de pocos días, por lo que decidimos utilizar la suite data integration de Pentaho (pdi), dado que es totalmente free y conocida, al ser la utilizada en el curso de Diseño y Construcción de Datawarehouse de la Facultad de Ingeniería.

Más en detalle, utilizamos la herramienta GeoKettle [43] de Pentaho para los procesos ETL. En primer lugar precargamos las tablas de tipo de crimen, origen, tiempo y ubicación geográfica.
Para cargar la tabla de ubicación geográfica, previamente cargamos las tablas de departamentos, localidades y barrios, obteniendo los shapefiles correspondientes del Catálogo de Datos Abierto [44]. Para exportarlos en la base de datos PostGIS usamos la herramienta QGIS [45], la cual permite cambiar fácilmente los sistemas de referencia de los shapefiles e importarlos a la base de datos de interés.

Luego cargamos la tabla de tiempo, generando fechas en un rango de X cantidad de años a partir de una fecha de inicio.
También agregamos los datos en la tabla de tipo de crimen y origen, teniendo en cuenta los datos de la base de datos relacional.

En la herramienta GeoKettle se definen transformaciones con pasos los cuales pueden ser input de entrada, filtros, cálculos, procesamiento con código JavaScript, entre otros. La entrada de la transformación principal es la tabla de crímenes. Se asocia con la fecha y hora correspondiente, el tipo de crimen y origen. Para la ubicación, dado que tenemos la latitud y longitud del punto, se tiene que ejecutar una consulta a la base de datos para saber a qué barrio pertenece dicho punto. Esto se hace mediante una consulta sql a la base de datos geográfica intersectando el punto con las distintas áreas para ver adentro de cual está.

El servidor OLAP que ofrece Pentaho es Mondrian, un servidor del tipo ROLAP (OLAP relacional). Los modelos multidimensionales que son interpretados por Mondrian consisten en archivos XML denominados Schemas, los cuales tienen un formato determinado. En ellos se definen los cubos, dimensiones, niveles, jerarquías, etc. sobre los que se trabajará y realizarán análisis. Existe una herramienta gráfica de escritorio para el diseño de Schemas denominada Schema Workbench [46].

Realizamos el schema desde dicha herramienta Schema Workbench, donde definimos el cubo con sus distintas dimensiones, y la medida cantidad.

Para la plataforma Business Intelligence utilizamos la versión 5.4 de Pentaho BI. El servidor de Pentaho se puede iniciar y detener manualmente mediante los scripts start-pentaho.bat y stop-pentaho.bat respectivamente, que se encuentran en el directorio de instalación. El servidor está configurado para correr sobre Apache Tomcat. Por defecto, Tomcat trabaja sobre el puerto 8080, por lo que, cuando se inicie el servidor de Pentaho, el mismo será accesible localmente a través de la página http://localhost:8080/pentaho. Se puede ingresar como usuario administrador, cuyo usuario y clave son admin y password respectivamente. También es posible ingresar como usuario de negocio, cuyo usuario y clave son suzy y password respectivamente. Luego del login como administrador, se muestra la página de bienvenida.

Luego de realizar el schema que define a los cubos de nuestra realidad, publicamos dicho schema en el servidor bi-server (previamente iniciado), que se encuentra en http://localhost:8080/pentaho con las credenciales correspondientes. A su vez desde la web especificada anteriormente hay que configurar el data source que se conecta a la base de datos del datawarehouse.

Visualizamos la información mediante distintas herramientas gráficas, entre las que se incluyen las proporcionadas por Pentaho (JPivot, CDE), y también realizamos análisis utilizando Saiku. Saiku permite la presentación de los datos en dashboards así como también en mapas.

5.1.2.3 Web de alertas

En este proyecto nos enfocamos en la capa de entrada e ingestión, así como también en el procesamiento de los datos. Igualmente surgió la necesidad de poder visualizar de alguna manera los datos que se generan en dicho procesamiento, por lo que decidimos realizar como parte del prototipo una web simple de alertas.
Se muestra en un mapa, en tiempo real, los crímenes ingresados a la topología Storm con su información correspondiente, de los últimos x minutos configurable.

La idea principal es que por cada crimen que entra en la topología Storm, se invoque a un web service que generará la alerta. El encargado de invocar al web service será el bolt AlertBolt. Luego, la aplicación que recibe el request tomará los datos y los expondrá en un mapa.

Por consiguiente implementamos una aplicación web Java utilizando las tecnologías JSF (Java Server Faces) para la interfaz de usuario y JAX-WS para los web services. También recurrimos a OpenLayers para el despliegue del mapa en la web, en su versión 2.

En un principio utilizamos a Google Maps como proveedor de mapas, a través de su API. Pero en Junio de 2018, la API de Google deja de ser gratuita, requiriendo un método de pago para poder emplearla. A partir de entonces los desarrolladores reciben 200 dólares de crédito gratuito mensual [47]. Pero esto implica que sí o sí hay que vincular una tarjeta de crédito, por lo que decidimos cambiar Google Maps y usar OpenStreetMap (OSM) [48].
OpenStreetMap (también conocido como OSM) es un proyecto colaborativo para crear mapas editables y libres. Los mapas se crean utilizando información geográfica capturada con dispositivos GPS móviles, ortofotografías y otras fuentes libres. Esta cartografía, tanto las imágenes creadas como los datos vectoriales almacenados en su base de datos, se distribuye bajo Licencia Abierta de Bases de Datos (en inglés ODBL).

El mapa muestra los distintos puntos que se distinguen según el tipo de crimen. Se eligen diferentes imágenes para cada tipo, de forma de permitir visualizar fácilmente qué tipos de crímenes son los que se están alertando, y a partir de esto decidir la prioridad que tienen, o la acción que se deba tomar. Se puede observar un ejemplo de este mapa en la figura 26.

Al hacer click sobre los puntos del mapa, se despliega un cuadro de diálogo con la información del mismo, entre los que se encuentran fecha y hora, descripción, calidad y origen.

Se da la posibilidad de agregar distintas capas al mapa base de OSM, siendo éstas las capas de localidades, barrios y departamentos del Uruguay. Para ello se utiliza el servidor de mapas Geoserver, el cual consta de un war que se despliega en un servidor de aplicaciones. Se accede a Geoserver y se configura la base de datos de la cual se va a acceder a los datos geográficos. Se publican las capas mencionadas y se definen estilos para las mismas.
Dichas capas se obtienen del Catálogo de Datos Abierto [44]. Las importamos a la base de datos geográfica mediante la herramienta QGIS.

Vale la pena de destacar que dicha web sirve para ver los crímenes que ocurren en el momento. Para ello al igual que lo hicimos en la topología, tenemos un Manager encargado de mantener una estructura de crímenes que se actualiza cada x minutos, configurable.

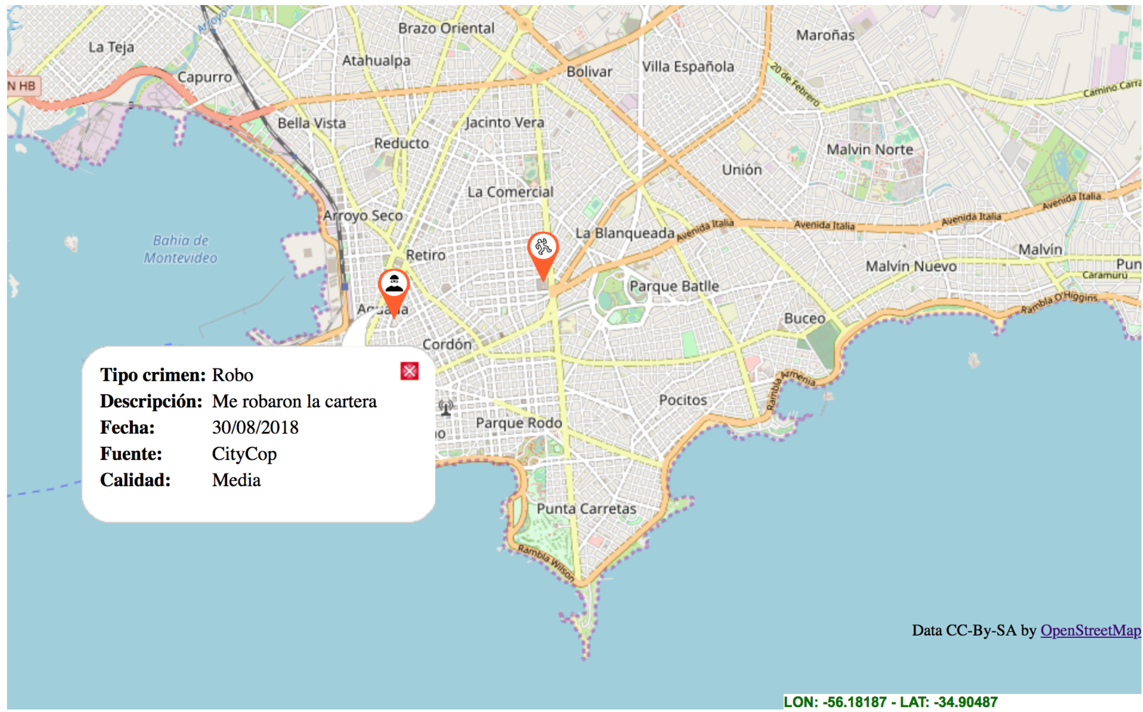


Fig. 26: Mapa de web de alertas

5.2 Herramientas utilizadas

5.2.1 Java

Java se define como un lenguaje de programación y una plataforma informática que fue comercializada por primera vez en 1995 por Sun Microsystems. Se puede utilizar como base para todo tipo de aplicaciones que van desde aplicaciones de red, aplicaciones móviles y embebidas, juegos, contenido web o software empresarial.

En la actualidad es uno de los lenguajes más utilizados. Java es un lenguaje orientado a objetos de alto nivel, y una de las principales características es que se trata de un lenguaje independiente de la plataforma, es decir, cualquier programa desarrollado en Java puede funcionar correctamente en computadoras de distintos tipos y con sistemas operativos diversos. Por lo tanto, facilita el trabajo de los programadores ya que no es necesario crear programas diferentes que se adapten a Windows, Linux, Mac, etc.

En nuestro caso nos resultó realmente útil ya que para el desarrollo contábamos con dos notebooks, una con sistema operativo Mac y otra con Windows, y realmente no tuvimos dificultades de compatibilidad en el código que estuvimos desarrollando. Ni tampoco con las librerías que utilizamos.

En Java todo el código fuente se escribe en archivos de texto plano que terminan en la extensión .java. Esos archivos luego son compilados por el compilador javac lo que retorna archivos .class, los que no contienen código nativo del procesador, en su lugar, contienen bytecodes: el lenguaje de máquina de Java Virtual Machine (JVM). Luego esa aplicación se ejecuta en una instancia de JVM. [49]

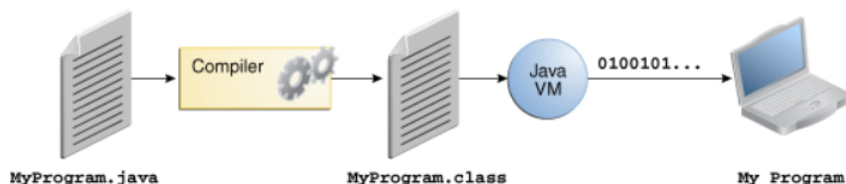


Fig. 27: Proceso de desarrollo con Java

Java ha sufrido cambios a lo largo de la historia. Además, en distintos momentos han coexistido distintas versiones o distribuciones de java con distintos fines. En la actualidad el Java vigente se denomina Java 2 o hay 3 distribuciones principales.

- J2SE o Java SE: Java Standard Edition, dirigido a desarrollo de aplicaciones cliente servidor. No incluye soporte a tecnologías Web. Es la base para otras distribuciones.
- J2EE: Java Enterprise Edition, orientado al ambiente empresarial y a la integración de sistemas. Incluye soporte para tecnologías Web. Su base es J2SE.
- J2ME: Java Micro Edition, orientado a pequeños dispositivos móviles(smartphones, tablets, etc).

5.2.2 Apache Maven

Maven es una herramienta Open Source que se puede utilizar para crear y administrar cualquier tipo de proyecto basado en Java.
Su principal objetivo es permitir a los desarrolladores comprender el estado completo del desarrollo de un proyecto en el período de tiempo más corto. Como herramienta de gestión de ciclo de vida de proyectos, Maven funciona en torno a fases y no como otras herramientas que se manejan por tareas. Maneja todas las fases del ciclo de vida de los proyectos, desde la validación, generación de código, planificación, pruebas hasta empaquetamiento, pruebas de integración, de verificación,

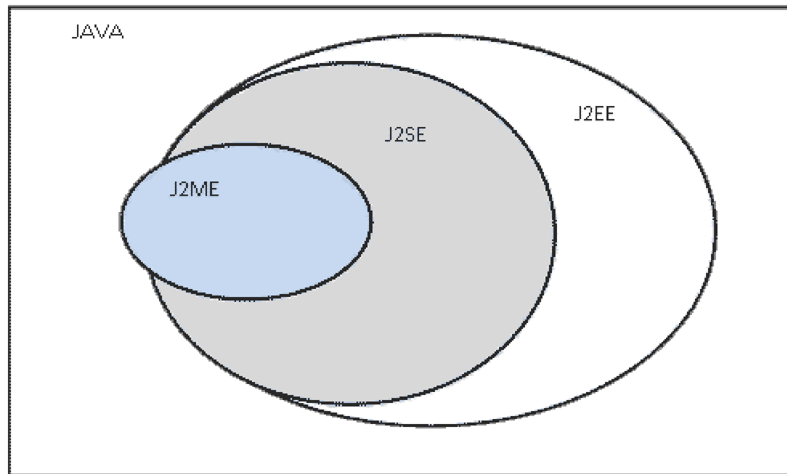


Fig. 28: Distribuciones Java

instalación, despliegue y creación e implementación de sitios.

En nuestro caso la utilidad más grande que le dimos a Maven es como herramienta de gestión de dependencias. Este es uno de los recursos más útiles que tiene Maven, simplemente es necesario definir las bibliotecas de las cuales depende la aplicación y Maven las localiza (o bien en el repositorio local o en el centralizado), las descarga y las utiliza para compilar el código. [50]

5.2.3 Eclipse

El proyecto Eclipse fue originalmente creado por IBM en noviembre de 2001, es una plataforma de desarrollo de código abierto diseñada para ser extendida de forma indefinida a través de plugins. Desde sus orígenes fue concebida para convertirse en una plataforma de integración de herramientas de desarrollo.

A su vez, Eclipse es también una comunidad de usuarios (Eclipse Fundation), que se encarga de extender continuamente las áreas de aplicaciones cubiertas. La fundación Eclipse es una organización sin fines de lucro que fomenta la comunidad de código abierto.
Para nuestro desarrollo utilizamos la versión 4.6 de Eclipse (Neon), versión disponible a partir de Junio de 2016.

5.2.4 JSF

JSF (Java Server Faces) es un framework basado en el patrón MVC (Modelo Vista Controlador) para aplicaciones Java basadas en web, el cual simplifica el desarrollo de interfaces de usuario en aplicaciones Java EE.

Facilita la construcción de interfaces de usuario (UI) para aplicaciones basadas en servidor, que usan componentes UI reutilizables en una página. La especificación JSF define un conjunto de componentes de interfaz de usuario estándar y proporciona una interfaz de programación de aplicaciones (API) para desarrollar componentes, y permite la reutilización y la extensión de los componentes existentes. [51]

5.2.5 OpenLayers

OpenLayers es una librería de código abierto que facilita la colocación de un mapa dinámico en cualquier página web. Puede mostrar mosaicos de mapas, datos vectoriales y marcadores cargados desde cualquier fuente.
OpenLayers ha sido desarrollado para promover el uso de información geográfica de todo tipo. Es completamente gratuito, JavaScript de código abierto, y publicado bajo la licencia FreeBSD. [52]

5.2.6 GeoServer

Es un servidor de mapas libre, escrito en Java, que corre sobre un web container Java EE. Es la implementación de referencia de WMS, WFS y WCS. Además posee una interfaz de usuario Web para realizar toda la configuración del servidor y permite configurar estilos mediante el estándar SLD (Styled Descriptor Language) de OGC. [53]

5.2.7 WFS

WFS (Web Feature Service) es un estándar de Web Service que soporta los métodos GET y POST. Define un protocolo para consultar y modificar información geográfica en GML y además provee algunas propias: DescribeFeatureType, GetFeature, GetGmlObject, Transaction, LockFeature (los WFS se clasifican según qué conjunto de las operaciones anteriores provee).

5.2.8 SoapUI

SoapUI es una herramienta open source de pruebas para SOAP y REST. Cuenta con una interfaz gráfica amigable que permite crear rápidamente pruebas funcionales, pruebas de regresión o pruebas de carga.[54]

En la actualidad es la herramienta líder a nivel global que te ofrece estas funcionalidades. Entre las características de SoapUI se encuentran:

- Testing funcional.
- Creación de pruebas Drag and Drop.
- Permite probar escenarios complejos.
- Simulación de servicios.
- Creación de test automático a partir de un WSDL.
- Pruebas de seguridad.
- Pruebas de carga.
- Validación de performance.

5.2.9 Bases de datos

5.2.9.1 PostgreSQL

PostgreSQL [55] un potente sistema de base de datos relacional de código abierto. Cuenta con más de 20 años de desarrollo activo y una arquitectura probada que se ha ganado una sólida reputación de fiabilidad e integridad de datos. Se ejecuta en los principales sistemas operativos que existen en la actualidad.Es totalmente compatible con ACID, tiene soporte completo para claves foráneas, uniones, vistas, triggers y procedimientos almacenados (en varios lenguajes).

5.2.9.2 PostGIS

PostGIS [56] es un software compatible con Open Geospatial Consortium (OGC) utilizado como extensor para PostgreSQL. Es open source y gratuito.

PostGIS agrega funciones y tipos geoespaciales adicionales que hacen que el manejo de datos espaciales en la base de datos sea más fácil y potente. El lenguaje es similar a SQL y permite el análisis espacial y las consultas típicas que se realizarán en datos espaciales con relativa facilidad. Esto lo convierte en un backend relativamente poderoso para las bases de datos dentro de un software más grande, ayudando a los proyectos a utilizar la funcionalidad similar a SQL para hacer análisis y consultas espaciales más complejos.

5.2.9.3 MongoDB

MongoDB [57] es un sistema de base de datos multiplataforma orientado a documentos, de esquema libre. Esto quiere decir que en lugar de guardar los datos en registros, guarda los datos en documentos los cuales son almacenados en BSON, que es una representación binaria de JSON. Una de las diferencias más importantes con respecto a las bases de datos relacionales, es que no es necesario seguir un esquema. Los documentos de una misma colección (concepto similar a una tabla de una base de datos relacional) pueden tener esquemas diferentes.

5.2.10 Kettle

Es una herramienta ETL de la suite open source Pentaho Data Integration. Ofrece la interfaz necesaria para extraer datos de diferentes tipos de fuente (archivos CSV, Excel, bases de datos, archivos xml, entre otros), realizarles modificaciones, filtros y otras transformaciones a los datos, y cargarlos en archivos o bases de datos destino.

A cada acción se le llama “step”; el proceso de Extracción-Transformación-Carga es una secuencia de steps que recibe el nombre de transformación, y la secuencia de diversas transformaciones (“programa principal” de Kettle) recibe el nombre de “Job”. En la práctica se observa que cada dato cambia de un step al siguiente de forma independiente al resto de los datos. Por lo que los registros se pueden desordenar a lo largo de la transformación y del job. [58]

5.2.11 GeoKettle

GeoKettle [43] soporta las principales bases de datos relacionales (Oracle, PostgreSql, PostGis, MySql). Se diferencia con la versión original de Kettle en que agrega un nuevo tipo de dato geométrico, e incluye nuevos “steps” que permiten unir sistemas de proyección cartográfica con campos geométricos, como ser:

- GIS File Input: Soporta la lectura de archivos de datos GIS. Por ahora solo soporta Shapefiles.
- GIS File Output: Soporta la escritura en archivos Shapefiles.
- Soporte de SRS (Sistemas de referencia espacial)
- Set SRS: Establece un SRS.
- Transformación SRS: Transforma las coordenadas (reproyección) de geometrías de un SRS a otro.

El traspaso de transformaciones de Kettle a GeoKettle no es directo. Es decir, transformaciones que funcionaban correctamente en Kettle pueden requerir algunos ajustes para funcionar en GeoKettle.

5.2.12 Mondrian

El núcleo de Mondrian es un JAR que actúa como "JDBC para OLAP": facilitando conexiones y ejecutando consultas SQL contra la base de datos relacional que sirve los datos. Los binarios de Mondrian vienen empaquetados de diferentes maneras:

- Como un paquete WAR que contiene Jpivot, un framework para trabajar con aplicaciones web y tecnología OLAP, junto con un ejemplo de datos que pueden ser cargados en una base de datos.
- Como un paquete WAR que además de contener a Jpivot, incluye una base de datos Derby, con lo que no se requiere ninguna configuración extra, aparte del despliegue sobre el servidor de aplicaciones.

Es un servidor ROLAP (Relational Online Analytical Processing). Se basa en Java y se encuentra en el dominio de creación de informes y almacenamiento de datos. Es útil cuando se utiliza para análisis que involucra la extracción de datos. Es un servidor de código abierto OLAP y admite el lenguaje de consulta MDX (Expresión multidimensional). El esquema de Mondrian es un descriptor de metadatos universales compatible con casi cualquier herramienta de cliente OLAP. [59]

5.3 Pruebas del prototipo

Se realizaron diferentes pruebas de concepto a lo largo del proceso de implementación.
Dado que tenemos tres componentes principales, los cuales pueden ser mapeados a proyectos en nuestro IDE Eclipse, primero hicimos una verificación de cada componente. Luego se realizaron test de integración y carga de los componentes en conjunto.

5.3.1 Pruebas de integración y carga

Para realizar estas pruebas utilizamos la herramienta SoapUI encargada de invocar el web service expuesto por el proyecto crime-integration. Dicho web service fue publicado de manera local utilizando el servidor Apache Tomcat, y por último la web de alertas también se publicó pero en este caso utilizando WildFly.
El objetivo principal de las mismas era probar los distintos flujos de la topología de Storm además de las conexiones entre los componentes (Kafka, Twitter4j, web de alertas).

5.3.1.1 Integración con Twitter

El primer flujo que se prueba por completo fue la integración con Twitter. En este caso se configuró la API Twitter4j para que llegaran a la aplicación los tweets de uno de los integrantes del equipo @FacundoAguero09. Luego publicábamos tweets desde esa cuenta o arrobabamos a esa cuenta para que los tweets ingresaran a la aplicación, a su vez realizábamos publicaciones en las que el contenido tenga alguna de las palabras claves que buscamos para poder clasificar la denuncia (palabras definidas en el diccionario).
Para poder obtener la ubicación geográfica del tweet es necesario activar la ubicación precisa desde la app de Twitter, como se puede ver en la figura 29.
Luego corroboramos que el crimen obtenido mediante el tweet se guarde de forma correcta, y además se encuentre clasificado de acuerdo con el contenido. En los casos que se genera una alerta en la web corroboramos que llegara, se ubique en el lugar correcto desde donde se había originado el tweet.

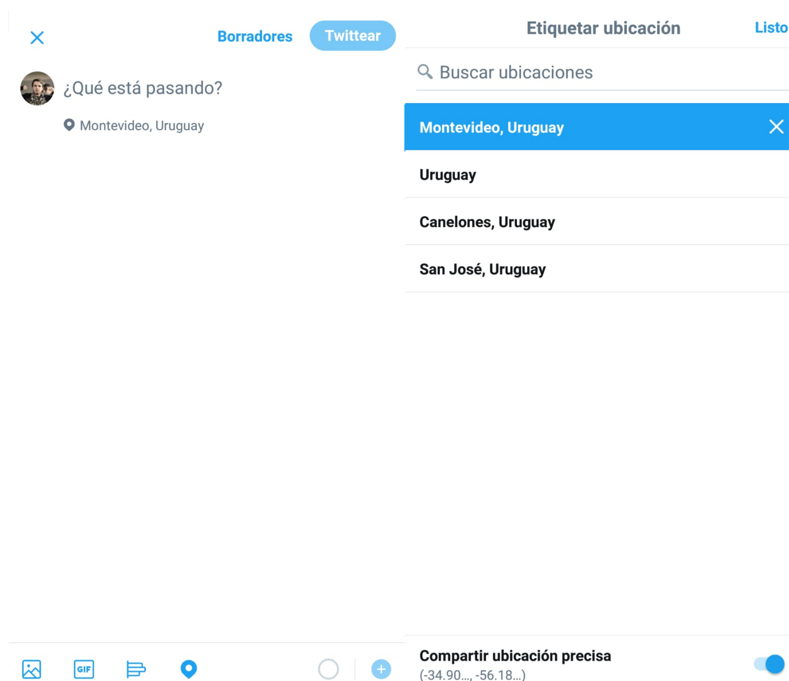


Fig. 29: Twitter: Ubicación precisa

5.3.1.2 *Integración con WS y Kafka*

Para probar el proyecto crime-integration se realizó la publicación del mismo de manera local en un servidor Tomcat, y a través de la herramienta SoapUI se consume el Web Service con distintos datos. Un ejemplo de una invocación utilizando SoapUI se muestra a continuación:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ser="http://services">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:insertCrime>
         <ser:crime>
            <ser:crimeType>Robo</ser:crimeType>
            <ser:description>descripcion</ser:description>
             <ser:latitude>-34.89640408</ser:latitude>
            <ser:longitude>-56.1895629</ser:longitude>
            <ser:origin>CityCop</ser:origin>
            <ser:place>Lugar</ser:place>
            <ser:quality>1</ser:quality>
            <ser:target>Persona</ser:target>
         </ser:crime>
      </ser:insertCrime>
   </soapenv:Body>
</soapenv:Envelope>
```

Realizamos las mismas validaciones que las pruebas de Twitter, esto es que quedaran guardados los datos en las bases de datos correspondientes, y que se generaran alertas en la web cuando correspondía.

5.3.1.3 *Pruebas de carga*

Apache Storm es una herramienta de procesamiento de datos en tiempo real, por lo tanto se hicieron pruebas de carga para observar el comportamiento de la topología, al tener que procesar una gran cantidad de datos que llegan de manera constante. Para el mismo se utilizo el test suite de SoapUI, en particular la API de Load Test, la cual se puede ver un ejemplo en la figura 30.

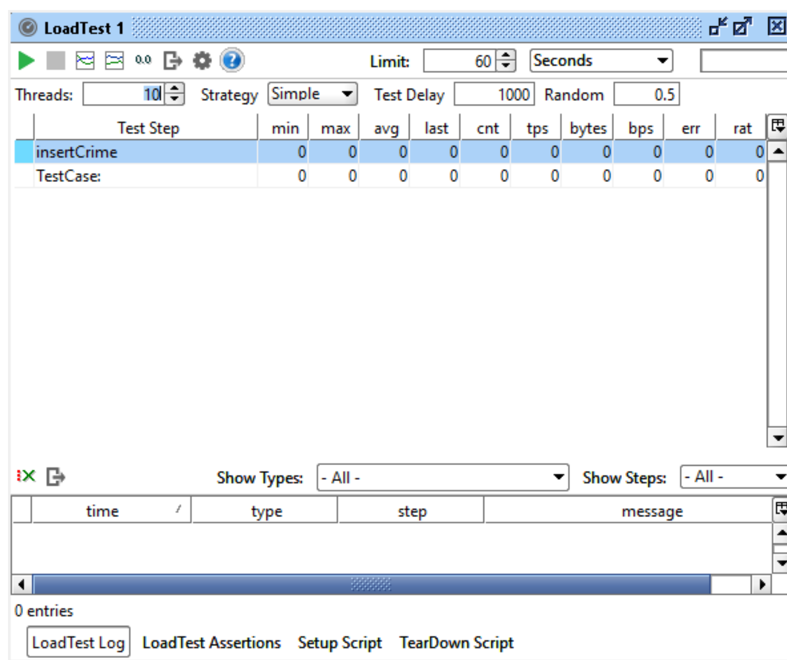


Fig. 30: Pruebas de carga

En esta caso está configurado para consumir la operación insertCrime del web service crime-integration, durante 60 segundos.
La cantidad de hilos indica cuántos clientes se simulan que consuman la operación, en el ejemplo la cantidad de hilos es 10. Para introducir el tiempo de espera entre una petición y otra se tiene el campo Test Delay, el cual en la figura 30 está en 1000 milisegundos. Para las pruebas se puso en 0 para que se generen peticiones constantemente.

Creamos un test de carga y lo probamos con diferentes configuraciones de los componentes de Storm. Las configuraciones que se modificaron para ver el comportamiento fueron: la cantidad de workers de la topología, y cantidad de executors y task para cada componente (bolt, spout).

- 1 worker, 1 executor con una task por componente.
- 1 worker, 1 executor con dos tasks por componente.
- 1 worker, 2 execuros con una task por componente.
- 2 workers, 1 executor con una task por componente.
- 2 workers, 1 executor con dos tasks por componente.
- 2 workers, 2 executor con una task por componente.
- 2 workers, 4 executors con una task por componente.
- 2 workers, 10 executors con una task por componente.

Estas pruebas se hicieron en una notebook localmente, no con la topología levantada en el cluster, por lo tanto hay que ser prudente a la hora de ver los resultados.

Observamos que al aumentar un worker en la topología no se produjo cambio en la velocidad de procesamiento, esto se da porque los dos workers ejecutan en la misma PC, entonces no se agrega paralelismo a la solución.Lo mismo sucede al agregarle más tasks a los executors, no hacen que el procesamiento sea más rápido.
Por otro lado, sí se nota el cambio al agregar mayor cantidad de executors de los componentes; al tener más instancias de los componentes levantados hay mayor paralelismo y se procesan más cantidad de tuplas a la vez.
En la figura 31 se ve una prueba de carga completa, 10 threads consumiendo la operación insertCrime de manera continua durante 60 segundos.
Se pueden ver más detalles en el Anexo de pruebas.

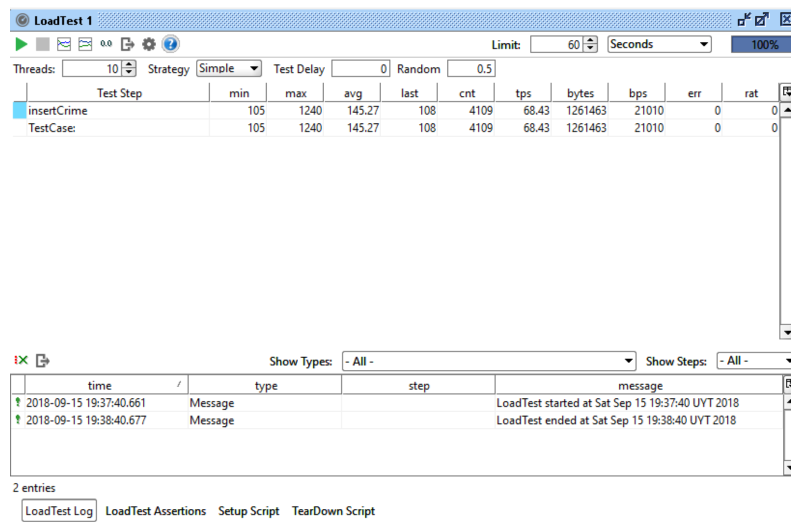


Fig. 31: Pruebas de carga completa

6 Gestión del proyecto

El siguiente capítulo tiene como objetivo mostrar la planificación inicial del proyecto, así como también las diferentes desviaciones que se sufrieron a lo largo del mismo. Se presentan diagramas de Gantt representando tanto la planificación inicial como la final, y se detallan las distintas etapas del proyecto.
Asimismo se muestran las metodologías y herramientas utilizadas en el desarrollo del mismo.

6.1 Gestión

El presente proyecto comienza en Abril de 2017 en el marco de la asignatura “Proyecto de Grado” de la Facultad de Ingeniería, Universidad de la República. Comenzamos con una reunión inicial con la tutora, donde planteamos los objetivos principales del proyecto así como también el alcance del mismo. Se nos proporcionó bibliografía de interés y una copia de los artículos en los que nos basamos.
Luego realizamos la planificación de las diferentes etapas que componen el Proyecto de Grado, el cual se puede visualizar en el diagrama de Gantt de la figura 32.
El proyecto inicial sufrió desviaciones, quedando finalmente las etapas mostradas en las figuras 33 y 34.

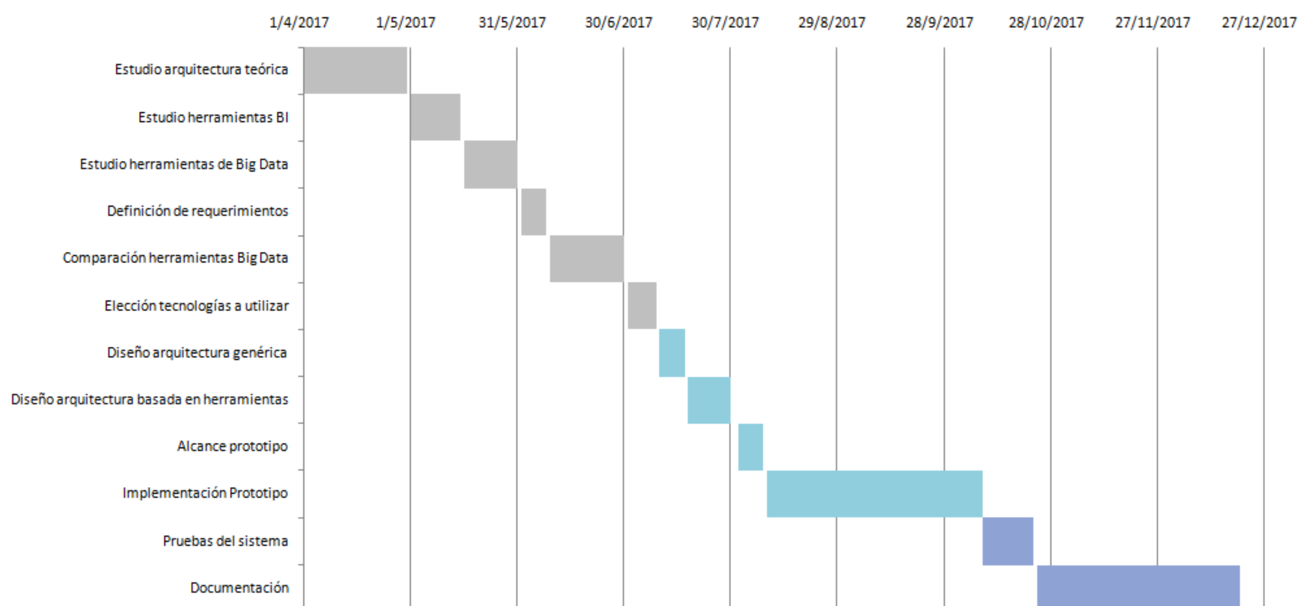


Fig. 32: Diagrama de Gantt inicial

6.1.1 Etapas

A continuación se detallan las distintas etapas del proyecto.

6.1.1.1 Estudio arquitectura teórica

El proyecto comienza con un análisis exhaustivo de los artículos de referencia, centrándonos en el estudio de la Arquitectura teórica propuesta en el artículo "GeoBI and Big VGI for Crime Analysis and Report" [1], dado que es el puntapié inicial para nuestro proyecto. Se investiga cada capa de la misma y se documenta al respecto.

6.1.1.2 Estudio herramientas BI

En un principio dado que la arquitectura tiene un gran componente de Business Intelligence, investigamos las distintas herramientas que existen en el mercado, decidiendo implementar un

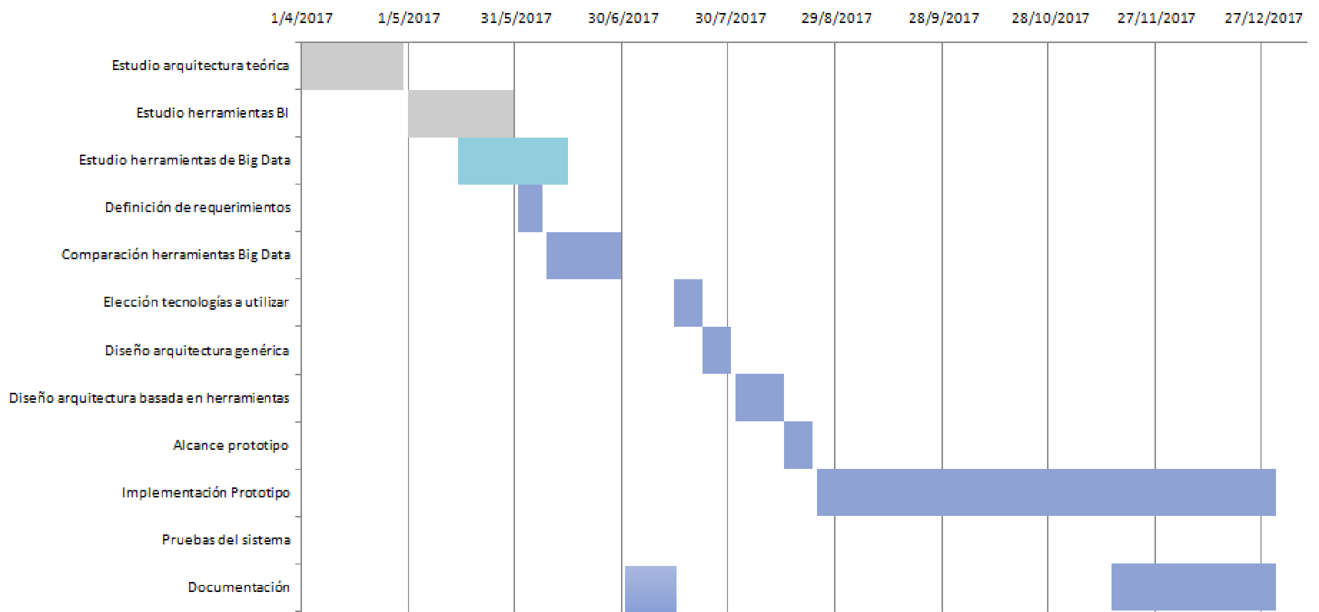


Fig. 33: Diagrama de Gantt final 1

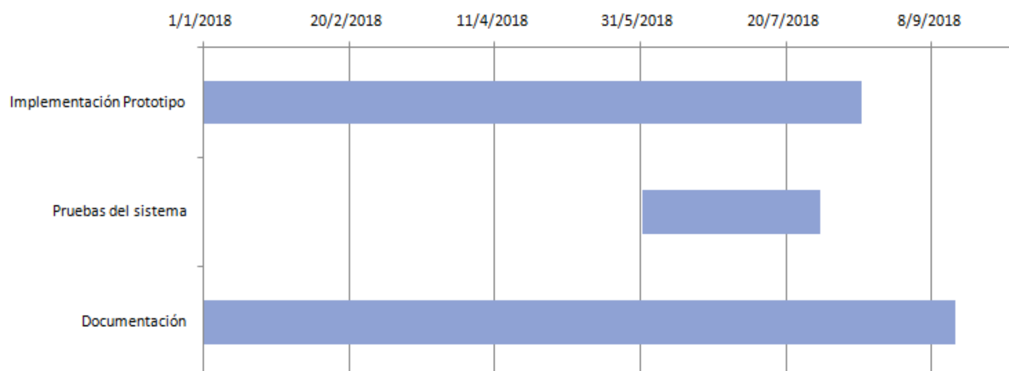


Fig. 34: Diagrama de Gantt final 2

pequeño prototipo para cada una de ellas, que tenga la misma funcionalidad. A finales del primer bimestre, luego de una reunión de avance, se cambia el enfoque del proyecto y este estudio de herramientas pasa a tener menor prioridad.

6.1.1.3 Estudio herramientas Big Data

Estamos ante un problema de Big Data, el cual es un término relativamente nuevo y desconocido para nosotros. Se tiene que aprender de qué se trata un problema/solución de Big Data, así como también analizar las herramientas que se puedan encontrar. En primer lugar investigamos las herramietnas existentes, seleccionando algunas de ellas para su posterior estudio en detalle. Luego realizamos un estudio teórico de las herramientas seleccionadas.

6.1.1.4 Definición de requerimientos

Posteriormente a entender la arquitectura teórica y relevar las herramientas, se definen cuáles serán los requerimientos. Es decir, determinamos qué es lo que pretendemos lograr con este trabajo, llegando a distinguir los siguientes hitos: diseñar una arquitectura de componentes identificando

los productos a usar, y diseñar un prototipo de análisis de información de crímenes, el cual es un problema de Big Data con procesamiento en tiempo real.

6.1.1.5 Comparación herramientas Big Data

En esta etapa hicimos un análisis comparativo de las herramientas de Big Data investigadas anteriormente. Seleccionamos las características que son relevantes para nuestra solución, y vimos como se comportan cada una de ellas.

6.1.1.6 Diseño arquitectura genérica

Dado que nos enfocamos en la capa de ingestión y procesamiento, y en una solución en tiempo real, diseñamos una arquitectura genérica para abordar dicho problema.

6.1.1.7 Diseño arquitectura basada en herramientas

A partir de la arquitectura teórica y la definida anteriormente, se diseña una arquitectura basada en productos específicos, donde se acota dicha arquitectura y se especifican los componentes y las herramientas principales de cada capa.

6.1.1.8 Alcance prototipo

Dada la arquitectura diseñada anteriormente se define el alcance del prototipo, es decir que módulos se harán y que herramientas se utilizarán.
Se diseña la topología de Apache Storm, así como las bases de datos correspondientes.

6.1.1.9 Implementación del Prototipo

La implementación del prototipo comienza con la creación de los spouts y bolts de la topología Storm, utilizando el lenguaje Java. Luego se realiza la integración con Twitter y el almacenamiento en base de datos. Posteriormente se agrega la lógica de procesamiento en cuanto a filtrado, validaciones, entre otros.
Se crea una aplicación que expone un web service y se comunica con Storm a través de Kafka. Esta integración implica más tiempo del estimado debido a incompatibilidad de versiones y falta de documentación.
Se desarrolla la lógica de cruzamiento de datos en busca de coincidencias. Se implementa un aplicación web de alertas en tiempo real que muestra un mapa con los crímenes recientes, así como una sistema de datawarehousing para el análisis posterior de los datos.

A principios de diciembre se realizó una presentación de avance con la tutora e invitados, donde se valida el trabajo realizado hasta ese momento.

6.1.1.10 Pruebas de sistema

Al finalizar la implementación realizamos pruebas a nivel local de todos los componentes de la plataforma, entre las que se encuentran la integración con Twitter, la integración con Kafka, y pruebas de carga para observar el comportamiento de Apache Storm.

6.1.1.11 Documentación

A lo largo del proceso, realizamos la documentación del proyecto para detallar la solución planteada. Contiene un marco teórico donde se definen los conceptos relevantes, el estudio de herramientas de Big Data, el diseño e implementación de la arquitectura, junto a su validación.

6.1.2 Desviaciones

La estimación inicial se vio afectada por diversos factores. Luego de realizar el estudio de la arquitectura teórica, decidimos centrarnos en una solución de BI, por lo que se le dio suma importancia a la investigación de herramientas de esta área, generando para cada una un pequeño prototipo de un requerimiento sencillo.

Sin embargo, a finales del primer bimestre presentamos el avance del proyecto a la tutora, profesores, y entre ellos Tatiana Delgado, coautora del artículo en el que se basa este proyecto. Nos propusieron cambiar el enfoque del mismo ya que hay componentes más importantes dentro de la arquitectura, que a su vez presentan un mayor desafío dada la complejidad, la importancia y que abarcan un área en la cual no hay mucha investigación, que está en desarrollo constante.
En consecuencia el estudio de herramientas de BI deja de ser elemental y el foco se pone en investigar las distintas herramientas de Big Data existentes.

A su vez, tanto Apache Storm como Apache Kafka eran herramientas desconocidas para nosotros, por lo que nos llevó más tiempo que el estimado el desarrollo de la topología y la integración con la cola de mensajes. Nos encontramos con un gran desafío al momento de realizar el cruzamiento de datos.
Kafka no deja claro en su documentación la configuración desde código Java, por lo que nos encontramos con algunos problemas mayormente de versionado.

El mapa de web de alertas se implementó en un principio usando a Google como capa base, pero luego en Junio de 2018, deja de ser libre por lo que se tiene que modificar dicho proyecto y cambiar la capa de Google por Open Street Map.

6.2 Metodología

6.2.1 Scrum

Scrum [60] es una metodología de desarrollo ágil. Se basa en aspectos como la flexibilidad en la adopción de cambios y nuevos requisitos durante un proyecto complejo, el factor humano, la colaboración e interacción con el cliente y el desarrollo iterativo como formas de asegurar buenos resultados.
En el presente proyecto adoptamos algunos aspectos de Scrum en la etapa de implementación para ayudarnos en la organización.

Dentro de un proyecto de Scrum se identifican los siguientes componentes:

- Product Backlog: es un listado ordenado de todo aquello que es necesario que forme parte del producto y es la única fuente de requerimientos o cambios a realizar sobre el producto. El Product Backlog se forma de ítems o características del producto a construir. Es importante que exista una clara priorización, ya que es esta priorización la que determinará el orden en el que el equipo de desarrollo transformará las características en un producto funcional acabado. Esta prioridad se define entre el equipo de desarrollo y el tutor del proyecto. Se debe aprovechar la construcción iterativa y evolutiva de Scrum para mitigar riesgos de forma implícita: construyendo primero aquellas características con mayor riesgo asociado y dejando las que poseen menor riesgo para etapas posteriores. A su vez los items prioritarios deben estar expresados con un nivel detalle mucho mayor que los items de menor prioridad.
- Sprint: Las iteraciones en Scrum se conocen como Sprints. Definimos sprints de dos semanas, siendo lo recomendado por la bibliografía y lo que mejor se adapta a nuestra experiencia.
- Sprint Backlog: es el conjunto de ítems del Product Backlog que fueron seleccionados para trabajar durante un determinado Sprint.
- Sprint Planning: al comienzo de cada Sprint realizamos un Sprint Planning para planificar el trabajo que se hará durante el mismo. A su vez definimos los objetivos a cumplir en el Sprint. Al finalizar cada Sprint se realiza una reunión de revisión del Sprint.

Durante el proyecto, en la etapa de implementación, decidimos realizar sprints de dos semanas comenzando los Martes, con reuniones los Lunes con la tutora.

6.3 Herramientas usadas

6.3.1 Pivotal Tracker

Pivotal Tracker [61] una herramienta de gestión de proyectos ágil para la colaboración en tiempo real en torno a un backlog compartido y priorizado. Allí creamos un proyecto con las tareas, el cual tiene las siguientes columnas:

- Done: Tareas finalizadas.
- Current iteration: Tareas que se están haciendo en el momento.
- Backlog: Tareas para hacer en el sprint actual. Gestionado en el comienzo de cada sprint.
- Icebox: Tareas pendientes.

Se pueden priorizar las tareas asignando puntos (1, 2, 3, 5 y 8) según el nivel de dificultad.

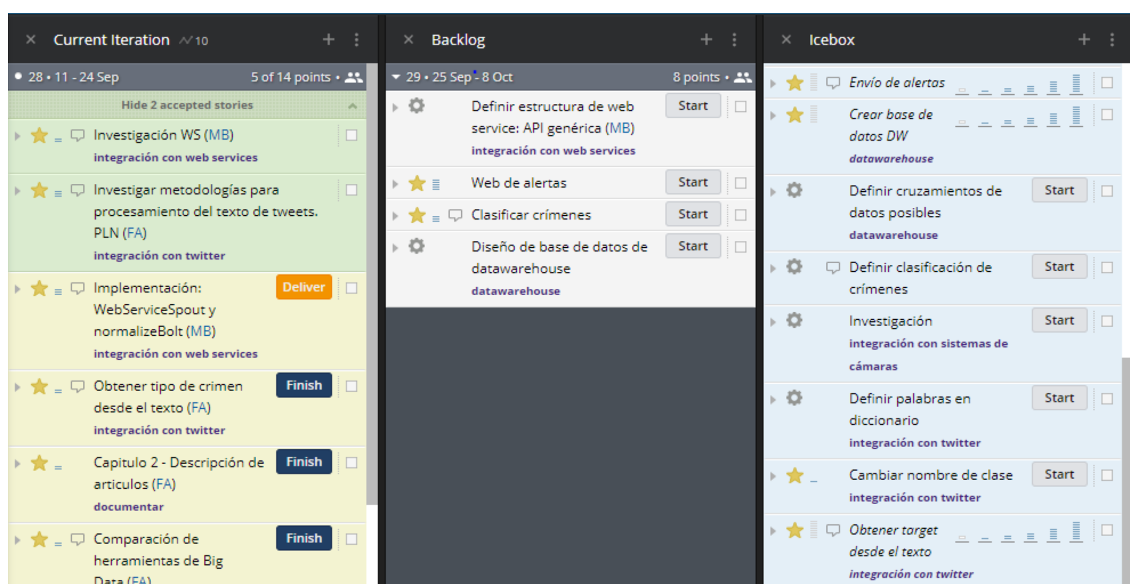


Fig. 35: Ejemplo Pivotal Tracker

6.3.2 Gitlab

Para la gestión del código de la aplicación usamos Gitlab [62], un gestor de repositorios Git basado en la web con wiki, seguimiento de problemas y características de canalización, utilizando una licencia de código abierto. Utilizamos el que proporciona la Facultad de Ingeniería.

6.3.3 Otros

Durante el proyecto compartimos documentos e información a través de Google Drive.
Para la comunicación entre los integrantes del grupo utilizamos Whatsapp.
El presente informe se escribió con LateX utilizando la herramienta online Overleaf [63], la cual permite la interacción en paralelo de los integrantes.

7 Conclusiones y trabajo a futuro

7.1 Conclusiones

Los objetivos principales del presente proyecto eran realizar un estudio y análisis de la arquitectura propuesta en los artículos de referencia [1][2], llevar a cabo un estudio de herramientas de Big Data, diseñar una arquitectura tecnológica basándonos en la utilización de productos específicos e implementar un prototipo de la misma.

Realizamos un estudio de la arquitectura propuesta en los artículos de referencia, lo que nos llevó a enfocarnos en la capa de ingestión y ETL, y a su vez a un problema de Big Data en tiempo real, dado que suponía un mayor desafío. Logramos cumplir con esta etapa.

En cuanto al estudio de herramientas de Big Data, nos enfocamos en una parte del ecosistema que consiste en las herramientas de procesamiento y las de mensajería. Llevamos a cabo un estudio comparativo para definir las herramientas que mejor se adaptaban a nuestro problema, y a partir de ahí definimos las que iban a formar parte de la arquitectura tecnológica. De esta parte podemos concluir que hay un ambiente muy variado de herramientas de Big Data, así como también existen varios stacks de trabajo ya definidos. La documentación para las mismas es extensa y no tuvimos mayores dificultades para encontrar información.

Definimos una arquitectura tecnológica a partir del estudio de herramientas, quedando nuestro stack de la siguiente manera: Apache Kafka + Apache Storm + MongoDB. A pesar de que también utilizamos otras tecnologías como ser PostgresSQL para el datawarehouse y web services.

En cuanto a la implementación del prototipo el resultado fue exitoso. Sin embargo se concluye que dado que las herramientas evolucionan constantemente surgen problemas de compatibilidad. Uno de los principales desafíos que tuvimos fue la integración entre Kafka y Storm, ya que hay versiones de los mismos que son incompatibles y no fue fácil darse cuenta que ese era el problema, dado que los errores que teníamos no eran muy descriptivos, y la documentación en este caso particular era escasa. Por momentos se evaluó la posibilidad de utilizar otra herramienta de mensajería, pero finalmente tuvimos éxito al integrarlas.

Sobre la utilización de Apache Storm para el procesamiento, es una herramienta que efectivamente se adapta a nuestros requerimientos, y además proporciona una solución escalable gracias a su arquitectura de componentes (Spouts y Bolts) la cual permite agregar nuevas fuentes de datos y modificaciones en el procesamiento de forma fácil. Asimismo permite crecer en capacidad de procesamiento de forma sencilla, agregando paralelismo a la solución. Esto último se puede hacer sin necesidad de modificaciones de código, ya que es parte de la configuración externa que agregamos.

Realizamos la implementación de una web de alertas en tiempo real que nos permitió la validación de la solución planteada, así como también un pequeño sistema de datawarehousing para la visualización del histórico de datos.

Durante el desarrollo del prototipo efectuamos pruebas de cada componente, y al finalizar se realizaron pruebas de integración como forma de probar el flujo completo del sistema, y también de carga para verificar el comportamiento y poder de procesamiento de Apache Storm.

A partir de esto concluimos que se lograron cumplir con los objetivos planteados al comienzo del proyecto, validamos la realización de la arquitectura propuesta en los artículos y adquirimos conocimientos del mundo de Big Data.

7.2 Trabajo a futuro

Los puntos que consideramos más importantes para continuar el desarrollo se describen a continuación:

- A la solución planteada de ingesta de datos y ETL se debería agregar la capacidad de procesamiento de datos de tipo batch, para obtener datos históricos de interés, por ejemplo la base de datos de CityCop, bases de datos del Ministerio del Interior u otra fuente de información.
- Al prototipo realizado, sería útil agregarle la integración con el proyecto de analíticas de video [23], el cual deja un log en un archivo de texto, del que se tendrían que obtener las alertas.
- En este último tiempo observamos que ha aumentado considerablemente tanto el uso como la popularidad de ciertas herramientas como ser Apache Flink. Por lo que resulta interesante implementar una solución similar utilizándola. Esto también permitiría hacer una comparación real con Apache Storm.
- Consideramos que sería de interés par los usuarios finales de nuestro sistema, contar con una aplicación móvil para recibir las alertas en tiempo real, de igual manera que la web de alertas.
- Culminar con la arquitectura propuesta, realizando un desarrollo que abarque el resto de las capas, para de esta manera validar por completo la solución planteada.
- Ampliar el rango de información que se obtiene y procesa de un crimen, por ejemplo obtener la víctima o la dirección del hecho a partir del contenido de un tweet.

Referencias

[1] Tatiana Delgado Federico Herrera, Raquel Sosa. GeoBI and Big VGI for Crime Analysis and Report. The International Conference on Open and Big Data (OBD-2015), 24-26 August 2015, Rome, Italy.

[2] Tatiana Delgado Federico Herrera, Raquel Sosa. Analíticas en línea de Big Data Espacial para análisis del crimen. 2015.

[3] RAE. Definición de crimen. `http://dle.rae.es/srv/fetch?id=BGTge4F`.

[4] Diario El País. Policía accede a cámaras de la imm. `https://www.elpais.com.uy/informacion/sociedad/policia-accede-camaras-imm.html`.

[5] CityCop. CityCop. `https://www.citycop.org/index.php?locale=ES\`.

[6] PNUD. Informe sobre Desarrollo Humano para América Central. 2009.

[7] Nathan Marz. *Big Data: Principles and best practices of scalable real-time data systems*. MANNING, 2015.

[8] Michael Stonebraker. Big data means at least three different things. `https://www.nist.gov/sites/default/files/documents/itl/ssd/is/NIST-stonebraker.pdf`.

[9] Guy Harrison. *Next Generation Databases*. Apress, 2015.

[10] José Ruiz Paradigma. Ecosistema Big Data. `https://www.paradigmadigital.com/ecosistema-big-data/`.

[11] Sinnexus. ¿Qué es Business Intelligence? `www.sinnexus.com/business_intelligence/`, .

[12] Sinnexus. Definición Datawarehouse. `https://www.sinnexus.com/business_intelligence/datawarehouse.aspx`, .

[13] Malinowski. *Advanced Data Warehouse Design*. Springer, 2008.

[14] OGC. Geobi. `http://www.opengeospatial.org/domain/geobi`.

[15] Michael F. Goodchilf. Citizens as Voluntary Sensors: Spatial Data Infrastructure in the World of Web 2.0. 2007.

[16] Base de datos relacional. `https://searchdatacenter.techtarget.com/es/definicion/Base-de-datos-relacional`.

[17] Medium. ¿Qué es un servicio web? `https://medium.com/grupo-carricay/qu%C3%A9-es-un-servicio-web-510be516863`.

[18] Machine Learning. `https://cleverdata.io/que-es-machine-learning-big-data/`.

[19] El Observador. CityCop: aplicación para denunciar delitos que levanta polvareda. `https://www.elobservador.com.uy/nota/citycop-aplicacion-para-denunciar-delitos-que-levanta-polvareda-201451618200`.

[20] Ministerio del Interior. Denuncias en línea. `https://denuncia.minterior.gub.uy/`, .

[21] Jordi Sabater Picañol Robert Serrat Morros. Big Data - Análisis de herramientas y soluciones. 2013.

[22] Ministerio del Interior. Observatorio Nacional Sobre Violencia y Criminalidad. `https://www.minterior.gub.uy/observatorio/`, .

[23] Silveira Chavat, Gómez. Algoritmos de inteligencia computacional para la deteccion de patrones de movimiento de personas. 2016.

[24] Apache hadoop. `https://hadoop.apache.org/`.

[25] Apache Hadoop YARN. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[26] Apache Spark. https://spark.apache.org, .

[27] Spark Streaming. https://spark.apache.org/streaming/, .

[28] Apache Flink. https://flink.apache.org/.

[29] Apache Samza. http://samza.apache.org/.

[30] Apache Storm. http://storm.apache.org/.

[31] Apache Kafka. https://kafka.apache.org/, .

[32] ZooKeeper. https://zookeeper.apache.org/doc/current/zookeeperOver.html.

[33] Jim Scott. Stream Processing Everywhere – What to Use? https://mapr.com/blog/stream-processing-everywhere-what-use/, .

[34] Jim Scott. Comparing Apache Spark, Storm, Flink and Samza stream processing engines. https://blog.scottlogic.com/2018/07/06/comparing-streaming-frameworks-pt1.html, .

[35] Apache Storm. Running Topologies on a Production Cluster. http://storm.apache.org/releases/1.1.2/Running-topologies-on-a-production-cluster.html.

[36] Twitter4j - a java library for the twitter api. http://twitter4j.org/en/index.html.

[37] Kafka vs JMS. http://cloudurable.com/blog/kafka-vs-jms/index.html, .

[38] Vishnu Viswanath. Realtime Processing using Storm-Kafka. http://vishnuviswanath.com/realtime-storm-kafka1.html.

[39] Kafka Producer and Consumer Examples Using Java. https://dzone.com/articles/kafka-producer-and-consumer-example, .

[40] JTS Topology Suite. https://www.osgeo.org/projects/jts/.

[41] GeoTools: The Open Source Java GIS Toolkit. http://geotools.org/, .

[42] Cuadrante mágico de gartner. https://rollupconsulting.com/cuadrante-magico-gartner-bi-2017/8.

[43] Geokettle. http://www.spatialytics.org/projects/geokettle/, .

[44] Catálogo de Datos Abierto. https://catalogodatos.gub.uy/dataset/limites-barrios.

[45] QGIS. https://www.qgis.org/es/site/.

[46] Schema Workbench. https://help.pentaho.com/Documentation/8.1/Products/Schema_Workbench.

[47] SmythSys. Cambios en la API de Google Maps. https://www.smythsys.es/10117/cambios-en-la-api-de-google-maps-hay-que-dar-la-tarjeta-y-nos-pueden-cobrar-a-los-desarrolla

[48] Open street map. https://www.openstreetmap.org/.

[49] Java. https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html.

[50] Maven. https://www.ibm.com/developerworks/ssa/library/j-5things13/index.html.

[51] Java Server Faces. https://www.tutorialspoint.com/jsf/.

[52] OpenLayers. https://live.osgeo.org/es/overview/openlayers_overview.html.

[53] Geoserver. http://docs.geoserver.org/latest/en/user/, .

[54] SoapUI. https://www.soapui.org/.

[55] PostgreSQL. https://www.postgresql.org/, .

[56] PostGIS. https://postgis.net/, .

[57] Bases de datos NoSQL, MongoDB y GIS. https://mappinggis.com/2014/07/mongodb-y-gis/, .

[58] Data Integration - Kettle. https://community.hitachivantara.com/docs/DOC-1009855-data-integration-kettle.

[59] Mondrian. https://mondrian.pentaho.com/documentation/installation_es.php, .

[60] Salias Alaimo. *Libro: Proyectos Ágiles Con Scrum*. Kleer.

[61] Pivotal Tracker. https://www.pivotaltracker.com.

[62] Gitlab: repositorio Facultad de Ingeniería. http://gitlab.fing.edu.uy/.

[63] Overleaf. https://www.overleaf.com/.

[64] Twitterid. https://tweeterid.com.

8 Anexo configuración

Para todas las configuraciones que se pueden hacer en el sistema se creó un archivo de configuración "default_config.properties". Utilizamos la clase Properties de Java para poder adquirir los valores de las propiedades y obtenerlas.
Las configuraciones que se pueden hacer en dicho archivo son:

- topology: El nombre de la topología.
- twitter.id: TwitterId de la cuenta de Twitter de la cual se van a obtener los tweets.
- kafka-spout: Nombre del Spout para la conexión con Kafka.
- kafka.zookeeper: Ip y puerto en el formato ip:puerto, del donde se encuentra Zookeeper.
- kafka.topic: nombre con el cual se identifica el topic en el cual se van a recibir los mensajes de Kafka (incoming en nuestro caso).
- kafka.zkRoot: ruta que utiliza el Spout de Kafka para almacenar información del consumo de mensajes.
- kafka.consumer.group: el Spout de kafka utiliza este atributo para armar la ruta en donde ir a buscar información sobre los mensajes.
- kafkaspout.count: -> Este se va.
- mongodb.host: host en donde se encuentra la base de datos MongoDB.
- mongodb.port: puerto en el cual conectarse con la base de datos MongoDB.
- mongodb.name: nombre de la base de datos MongoDB.
- postgres.jdbcUrl: URL para conectarse a la base de datos PostgreSQL a traves de JDBC.
- postgres.username: usuario para conectarse a la base de datos PostgreSQL.
- postgres.pass: contraseña del usuario postgres.username para conectarse con la base de datos PostgreSQL.
- num.of.workers: La cantidad de workers que se van a crear para la topología en el cluster.
- Los atributos: kafka.spout.parallelism.hint, twitter.spout.parallelism.hint, normalize.twitter.bolt.parallelism.hint, normalize.denuncias.bolt.parallelism.hint, classify.twitter.bolt.parallelism.hint, validate.bolt.parallelism.hint, classify.bolt.parallelism.hint, action.bolt.parallelism.hint register.bolt.parallelism.hint, alert.bolt.parallelism.hint, dashboard.bolt.parallelism.hin, add.struct.bolt.parallelism.hint
  Se utilizan para asignar a cada Spout y Bolt correspondiente el nivel de paralelismo, es decir cuantos executors de cada componente se van a instanciar en la topología.
- Los atributos: kafka.spout.tasks, twitter.spout.tasks, normalize.twitter.tasks, normalize.denuncias.tasks, classify.twitter.tasks, validate.bolt.tasks, classify.bolt.tasks, action.bolt.tasks, register.bolt.tasks, alert.bolt.tasks, dashboard.bolt.tasks, add.struct.bolt.tasks se utilizan para asignar la cantidad de threads en cada executor de los Spouts y Bolts.
- exclude.origin: lista de orígenes que se excluyen en el procesamiento.

9 Anexo Manual de Usuario

9.1 Herramientas

Para ejecutar el proyecto se requiere:

1. Apache Zookeeper 3.4.10
2. Apache kafka 2.9.1-0.8.2.1
3. MongoDB 3.4
4. PostgreSQL
5. Apache Tomcat 7
6. WildFly 8
7. Java

9.2 Zookeeper

Zookeeper tiene un archivo de configuración dentro de la carpeta conf, en el que se puede cambiar el puerto donde levanta, o la ruta en donde se guarda los logs (dataDir, clientPort).
Para levantar el Zookeeper se tiene que ejecutar el siguiente comando desde la carpeta base de Zookeeper:

```
$ bin/zkServer.sh start
```

En caso de usar Windows se tiene que ejecutar "zkServer.cmd".

9.3 Apache Kafka

Kafka tiene un archivo de configuración que se encuentra en la carpeta config, en este archivo se tiene que configurar los siguientes parámetros:

- port: el puerto en donde está escuchando kafka.
- log.dirs: la ruta en donde guarda archivos de logs y de estados. Puede ser una lista de archivos separados por coma.
- zookeeper.connect: IP y puerto en con el formato IP:puerto en donde está levantado Zookeeper. Es necesario para que pueda levantar Kafka que Zookeeper esté levantado.
- Hay más atributos de configuración que se pueden cambiar dependiendo de las necesidades, pero estos son los básicos para que pueda ejecutar con éxito.

Es necesario si ya no está creado, crear el topic para los mensajes de Kafka, a partir del comando:

```
$ bin/kafka-topics.sh --create --topic incoming --zookeeper localhost:2181 --partitions
    1 --replication-factor 1
```

De esta forma se crea el topic "incoming".

Para levantar Kafka se tiene que ejecutar el comando:

```
$ bin/kafka-server-start.sh server.properties
```

En donde server.properties es la ruta en donde se encuentra dicho archivo. Para el caso de Windows hay una carpeta "windows" dentro de la carpeta bin de Kafka en donde están los comandos para que levanten ahí. [38]

9.4 MongoDB

Arrancar el daemon de mongo ejecutando desde la carpeta de instalación el comando:

```
$ bin/mongod
```

9.5 PostgreSQL

Es necesario tener levantado el servidor de PostgresSQL, y cargar la base de datos con la estructura definida en el script de base de datos.

9.6 Crime-integration

Para poder enviar mensajes de a la cola de kafka es necesario tener publicado el Webservice crime-integration. Para esto es necesario tener Tomcat 7 con el proyecto publicado.
Es necesario configurar la IP y el puerto en donde esta escuchando Kafka BOOTSTRAP_SERVERS_CONFIG.

9.7 Alert-integration

El proyecto alert-integration se publica a través de un servidor WildFly 8, o bien desde eclipse o publicandolo fuera del mismo.

9.8 Storm

Lo primero que hay que hacer es modificar el archivo de configuración (default_config.properties) y poner los atributos correspondientes: IP y puerto zookeeper, topic de kafka, IP y puerto de la base de datos Mongo, atributos de la base de datos PostgreSQL(IP, puerto, nombre, nombre de usuario y contraseña). A su vez, se debe configurar la cantidad de workers, el nivel de paralelismo de los elementos y la cantidad de threads para cada executor de la topología. Para tener más información acerca de los atributos de configuración ver en la sección Anexo configuración.
Es necesario asignar el atributo de configuración twitter.id, en el cual va el TwitterId de la cuenta que se quiere obtener los mismo. Para obtenerlo hay que ir a una pagina web como twitterId.com[64], y a partir del nombre de usuario te devuelve el twitterId.
Por ejemplo el de la cuenta @Chorros_uy es 430737039.

10 Anexo pruebas

En esta anexo se describe los resultados de las pruebas sobre Storm, y como cambia el nivel de procesamiento con diferentes configuraciones. Cabe aclarar que las pruebas se realizaron en un notebook de manera local, lo que significa que los recursos disponibles para levantar la topología de Storm, Kafka, ZooKeeper, web services y bases de datos eran compartidos.

Las pruebas se hicieron realizando un test de carga de SoapUI, sobre el web service que recibe alertas. El mismo lo envía a la cola de Kafka, desde donde Storm lo obtiene. El test está configurado para realizar invocaciones de manera constante al web service durante un minuto.

#Workers	#Executors	#Tasks	Tiempo
1	1	1	2:20
1	1	2	2:23
1	2	1	1:32
2	1	1	2:18
2	1	2	2:19
2	2	1	1:30
2	4	1	1:13
2	10	1	1:06

Lo primero que podemos observar es que el tiempo que demora en procesar los mensajes, se reduce de manera significativa al aumentar la cantidad de executors por componente de Storm. Esto tiene sentido ya que aumenta la cantidad de instancias de los componentes, haciendo que se puedan procesar más mensajes de forma paralela.

El hecho de que no haya gran diferencia en el tiempo de procesamiento cuando se cambia la cantidad de workers se da porque esto no modifica la cantidad de componentes que se ejecutan, solo los distribuye de distinta manera. Esto y dado que se está ejecutando de manera local hace que no se mejore al aumentar la cantidad de workers. Lo mismo pasa con la cantidad de tasks por executor, no hace la diferencia por el mismo motivo, se ejecuta en una máquina con recursos muy limitados.

En la gráfica de la figura 36, se puede ver claramente como al aumentar la cantidad de instancias (Executor) de cada componente, el tiempo de procesamiento disminuye de manera significativa, tendiendo a un minuto. En ese caso sería que no hay latencia, ya que las pruebas se ejecutaban durante ese tiempo.
Para ser mas claro, a la derecha se muestra el tiempo de procesamiento y a la izquierda la cantidad de componentes de cada tipo.

Fig. 36: Workers, executors, tasks y tiempo de procesamiento