Workberch: Solución de Big Data para Workflows científicos

Equipo:
Martín Flores
Berch Gularte
Matías Fernández

Tutor: Guzmán Lambías

2015

Índice

1 Resumen

Los sistemas de workflows científicos son programas utilizados principalmente en la comunidad bioinformática para la experimentación in silico. Comúnmente, estos programas tienen que lidiar con grandes volúmenes de datos de diversos tipos y complejidades.

Por otro lado, en el mundo de IT surgen herramientas catalogadas como herramientas de Big Data. Estas herramientas surgen de la necesidad de tener que trabajar de forma eficiente con grandes y complejos volúmenes de datos.

Este proyecto se encarga de estudiar si es posible mejorar la performance de los sistemas de workflows científicos (SWfMS) utilizando tecnologías de Big Data y se plantea como objetivo, la construcción de un prototipo que mejore la performance de alguno de los SWfMS existentes.

Para esto se analizó el funcionamiento general de los SWfMS y se estudiaron dos herramientas específicamente: Taverna y Kepler. A su vez se hizo un análisis de los distintos tipos de herramientas de Big Data que existen para poder elegir aquellas que se adaptan mejor a la realidad de los SWfMS. De este análisis se optó por tomar a Taverna como SWfMS base y a Apache Storm como tecnología base de Big Data. Esta selección dio cómo paso a la construcción del prototipo denominado Workberch, el cual cubre las funcionalidades mínimas de Taverna pero suficientes para poder comparar su rendimiento.

Los resultados de las pruebas de performance demostraron que en algunos escenarios, el prototipo Workberch tiene menores tiempos de ejecución que Taverna, mientras que en los escenarios restantes, se plantean alternativas de mejora que permitirían evolucionarla y mitigar estas limitaciones.

Palabras claves

Workflows, workflows científicos, Big Data, paralelización, Taverna, Apache Storm, Play Framework, Redis.

2 Introducción

En este documento se detalla la investigación realizada, el análisis de la realidad planteada, y la solución propuesta, durante la ejecución del proyecto denominado “Workberch: Solución de Big Data para Workflows científicos” para el proyecto de grado en la carrera Ingeniería en Computación de la Facultad de Ingeniería - UDELAR.

2.1 Motivación

Una de las formas de experimentación realizadas en el mundo de la investigación científica, es la experimentación *In Silico.* Denominada de esta manera dado que el tipo de experimentación es llevada a cabo por una computadora, ya sea a través de cálculos o simulaciones.

En este nuevo contexto de experimentación los Workflow Management Systems (WfMS) juegan un papel crucial, contando con varias implementaciones en el mercado. Estos WfMS construidos para la experimentación científica son llamados Scientific Workflow Management Systems (SWfMS) y se inspiran en los WfMS empresariales de la industria, aunque manejan problemas particulares de la realidad del mundo científico.

Por otro lado en la industria IT se comienza a escuchar un término cada vez con más fuerza, el término Big Data. Este término refiere a la nueva tendencia en tecnologías para la manipulación, procesamiento y análisis de grandes volúmenes de datos.

Este tipo de herramientas emergentes pueden ser extremadamente beneficiosas para realidades donde el dato tiene un valor crucial, sobre todo si la generación y procesamiento de los mismos puede llegar a ser un problema con las tecnologías tradicionales.

Dado que en los SWfMS el dato y el procesamiento de los mismos es la principal tarea de estos sistemas, y además la realidad científica requiere que estos sistemas manejan y consulten grande cantidades de datos no triviales, la aplicación de tecnologías en esta realidad parece un caso de uso más que idóneo.

Es en la intersección de estas dos realidades donde el equipo trabajó para dar pautas de cómo estas realidades pueden ser combinadas, además de proveer una solución de SWfMS mejorada con tecnologías de Big Data.

2.2 Objetivo

El objetivo general de este proyecto es aplicar tecnologías de Big Data para mejorar la ejecución y recolección de datos de los SWfMS.

En particular para este proyecto se tomará Taverna como el SWfMS a investigar. Por el hecho de que los SWfMS son plataformas de ejecución, se hará especial hincapié en la investigación de herramientas que permitan ejecutar y manejar una gran cantidad de datos en un modelo de workflows.

Además de esto se determinará qué aspectos de Taverna serán mejorados o re implementados, también se decidirá los componentes a utilizar para mantener compatibilidad con la plataforma.

El resultado de la investigación es implementación de un prototipo utilizando las herramientas dadas cómo candidatas por el equipo.

2.3 Aportes del proyecto

Los principales aportes de este proyecto vienen dados por la implementación de un prototipo y la profunda investigación realizada para poder realizar el mismo. A continuación se da un pequeño breve punteo de este aporte.

- Investigación de distintos SWfMS comparándolos entre sí.
- Investigación y categorización de distintas herramientas de Big Data para la resolución de una gran variedad de problemas.
- Investigación de como algunas de estas herramientas son compatibles con la realidad de los workflows científicos.
- Implementación y diseño de un prototipo funcional fundado en las investigaciones previamente referidas.
- Realización de distintas pruebas para validar los atributos de calidad que se deseaban atacar.

2.4 Organización del documento

El documento se organiza en siete capítulos, de los cuales la presente *Introducción* es el primer capítulo.

En el segundo capítulo se da un *Marco Conceptual* que contiene toda la investigación realizada sobre los dos grandes temas de este proyecto, Workflows Científicos y Big Data. Este capítulo brinda todas las definiciones e ideas que se manejan en los capítulos siguientes.

En el tercer capítulo se encuentra el *Análisis, en* donde se presentan los requerimientos del proyecto luego de analizar algunas herramientas de Big Data que solucionarán algunos problemas de los workflows científicos. Además también se verán algunos problemas que se dan al utilizar herramientas de Big Data.

En el capítulo número cuatro se centra en el *Diseño* de la solución a implementar, presentando su arquitectura, los procesos y algoritmos que se necesitan implementar, y los atributos de calidad que se buscan atacar en la implementación del prototipo llamado Workberch.

En el quinto capítulo se ven algunos aspectos en cuanto a la implementación del prototipo y además se profundiza en las tecnologías que se utilizaron en la solución.

En el sexto capítulo se ve una pequeña reseña sobre la *Gestión del Proyecto.*

En el último y séptimo capítulo se ven la *Conclusión* del trabajo realizado y los trabajos a futuro para mejorar el prototipo Workberch luego de implementado el prototipo y habiendo ejecutado pruebas sobre el mismo.
.

3 Marco Conceptual

En este capítulo se busca introducir al lector en los dos temas principales que se tratan a lo largo del proyecto, Big Data y Workflows Científicos.

En la primera sección se profundiza en lo que es Big Data. Se analiza cómo y cuándo surge el término, y se busca cómo lo definen algunos de los principales actores de la industria.
Luego se construye una definición del término, que se enmarca en este proyecto.

En la segunda sección se aborda el otro tema a nivel macro, Workflows Científicos. Primero se da una breve definición de lo que se entiende por Workflows Científicos y se hace un relevamiento de los problemas que intentan resolver. También se identifican conceptos claves de este tipo de sistemas y se estudian dos sistemas de Workflows Científicos, Kepler y Taverna.

Por último se analizan que herramientas de Big Data son candidatas en el marco de este proyecto y se realiza un estudio de cómo combinar Workflows Científicos con Big Data.

3.1 Big Data

No es difícil realizar una búsqueda en internet y encontrar varios artículos que mencionan al término Big Data. El término ha captado la atención de los medios en el último tiempo y esto hace que se encuentren varias definiciones que no siempre están en sintonía.

Es por esto que en esta sección se intenta fijar la idea de lo que es Big Data y cómo se ubica en el contexto de este proyecto. Este proceso se realiza usando como punto de partida, los puntos de vista de alguno de los actores clave de la industria.

3.1.1 Definiciones

En el año 2001 el analista del META Group Inc. (ahora Gartner [1]) Douglas Laney [2] presentaba un informe [3] en el cual introducía el término Big Data. Lo primero que se puede leer del informe es el siguiente extracto:

> *"Las condiciones actuales de negocios y medios están empujando los principios tradicionales de gestión de datos a sus límites, esto da lugar a nuevos enfoques más formales."*

Este informe y las observaciones que se realizan son citados constantemente en artículos y definiciones de Big Data, por ejemplo en [4], [5] y [6]. Es por esto que la primera definición de Big Data que se estudia en este capítulo es la de Laney.

3.1.1.1 Laney

> *"Big data is high-volume, high-velocity and high-variety information assets that demand cost-effective, innovative forms of information processing for enhanced insight and decision making."*
>
> *Big Data son los activos "alto volumen", "alta velocidad" y "alta variedad" que demandan formas innovadoras y rentables del procesamiento de la información para mejorar su comprensión y la toma de decisiones." [7]*

En su definición Laney identifica tres dimensiones sobre la naturaleza de los datos en las cuales ve nuevas oportunidades. Las dimensiones que se identifican son; variedad, velocidad y volumen de datos, en la jerga se las conoce cómo **"las tres Ves"**. Se presenta una breve descripción de cada una de las dimensiones. [8] [9]

- **Variedad:**
  Hoy en día las organizaciones cuentan con múltiples fuentes de datos y cada una de estas con formatos y estructuras distintas. Se ha vuelto una necesidad poder relacionar los distintos tipos de datos (variedad) para poder generar nueva información.

- **Velocidad**:
  No solo se refiere al análisis en tiempo real de los datos, sino que también a la frecuencia con la que cambian. Detectar variaciones en la frecuencia con la que cambia un dato es un dato en sí.

- **Volumen:**
  Esta es la V más clara que refiere al tamaño de los datos. En estos tiempos se genera mucha información que tiene que ser almacenada en data sets gigantes para un pasado cercano.

3.1.1.2 Oracle

En junio de 2013 Oracle presenta el *white paper* "Oracle: Big Data for the Enterprise" [10], donde se presenta la visión que tiene Oracle en cuanto a Big Data y a su vez plantea un arquitectura para crear un stack tomando como base los productos y servicios que Oracle ofrece.

En el comienzo del *white paper* se realiza un estudio de los tipos de datos sobre los cuales Oracle entiende que Big Data aplica. Estos son:

- **Datos empresariales tradicionales:** Generados típicamente por CRM, ERP, Aplicaciones de comercio electrónico y sistemas contables.

- **Datos generados por sensores/máquinas:** Entre ellos se encuentran registros telefónicos, logs, sensores utilizados en procesos de manufactura.

- **Datos sociales:** Datos de servicio de clientes, redes sociales como Facebook y Twitter.

Utilizan las 3 Vs en el mismo sentido que la definición de Gartner y agregan una cuarta:

> *"Value. The economic value of different data varies significantly. Typically there is good information hidden amongst a larger body of non-traditional data; the challenge is identifying what is valuable and then transforming and extracting that data for analysis. "*
>
> *Valor. El valor económico de diferentes tipos de datos varía significativamente. Típicamente hay buena información oculta a lo largo de un conjunto no tradicional de datos; el desafío es identificar qué es de valor para poder extraer y transformar esos datos para el análisis.*

Luego el enfoque del documento está orientado a la infraestructura necesaria para construir sistemas acorde a los requerimientos de Big Data. Definen un modelo de capas para tratar con los requerimientos de Big Data con ejemplos de tecnologías concretas para aplicar en cada capa (por ejemplo bases de datos NoSQL, Hadoop, conectores de Oracle para Big Data). No se hizo un análisis de esta parte ya que se entiende que tiene un sesgo para utilizar el *stack* de tecnologías de Oracle.

3.1.1.3 Microsoft

El 11 de febrero de 2013 Microsoft en su centro de noticias publicaba el artículo The Big Bang: How the Big Data Explosion Is Changing the World [11]. Lo primero que se puede leer es el siguiente fragmento:

> *"REDMOND, Wash. – Feb. 11, 2012 – In the battle of the buzzwords, "big data" is about to render "guestimation" obsolete.*
>
> *This is big."*
>
> *REDMOND, Wash. – Feb. 11, 2012 – En la pelea por las palabras de moda "big data" está a punto de hacer la "especulación" obsoleta.*
>
> *Esto es grande.*

Más adelante dan una descripción concreta del término “Big Data”:

> *“Big data is the term increasingly used to describe the process of applying serious computing power – the latest in machine learning and artificial intelligence – to seriously massive and often highly complex sets of information.”*

> *Big data es el término cada vez más usado para describir el proceso de aplicar gran poder de procesamiento - lo último en aprendizaje automático e inteligencia artificial - a grandes y complejos volúmenes de datos*

Microsoft dice que ha habido un punto de inflexión como consecuencia de la alta proliferación, en la cantidad de datos, los avances en los algoritmos de aprendizaje automático y lo barato que resulta guardar gran cantidad de datos hoy en día. A su vez mencionan como el volumen y la no estructura de los datos son problemas con los que debe lidiar una solución de Big Data.

3.1.1.4 IBM

Un año antes que Oracle y Microsoft, el 18 de junio de 2012, IBM en su portal para desarrolladores publica el artículo “¿Qué es Big Data? Todos formamos parte de ese gran crecimiento de datos” [12]. En la introducción del artículo se puede leer:

> *“El primer cuestionamiento que posiblemente llegue a su mente en este momento es ¿Qué es Big Data y porqué se ha vuelto tan importante? pues bien, en términos generales podríamos referirnos como a la tendencia en el avance de la tecnología que ha abierto las puertas hacia un nuevo enfoque de entendimiento y toma de decisiones, la cual es utilizada para describir enormes cantidades de datos (estructurados, no estructurados y semiestructurados) que tomaría demasiado tiempo y sería muy costoso cargarlos a un base de datos relacional para su análisis. De tal manera que, el concepto de Big Data aplica para toda aquella información que no puede ser procesada o analizada utilizando procesos o herramientas tradicionales”.*

También se menciona explícitamente **“las tres Ves”** y se da una breve descripción de cada una de ellas, aunque al hablar sobre la velocidad solo se hace referencia a la velocidad de procesamiento.

Además se hace una categorización de los tipos de datos que aplican a Big Data, algo similar al que plantea Oracle, aunque agregan también datos biométricos (reconocimiento facial, genética) y los datos generados por humanos (emails, registros telefónicos, registros médicos).

Se realizan un breve relevamiento de las herramientas que se están usando en el momento y alguna de las líneas de investigación que ocurrían en el momento de escribir el artículo.

3.1.2 La definición del equipo

Cómo se ha visto a lo largo de las secciones anteriores, distintos agentes han hecho un esfuerzo por comunicar al mundo su definición de Big Data, ponderando en lo general sus productos. No ayuda tampoco que el término “esté de moda”. Por ejemplo al momento de buscar definiciones (mayo de 2014) uno de los resultados en la primera página del buscador de Google era un artículo [13] del portal web Puro Marketing con el título:

> *“¿Es realmente seguro el Big Data para usuarios y consumidores?*
>
> *¿Es Big Data el mayor peligro y amenaza para la seguridad y privacidad de los usuarios y consumidores?”*

En el artículo se trata a Big Data y Business Intelligence como si fueran lo mismo, lo cual no es cierto. Se menciona que la voracidad de las empresas por guardar más datos es consecuencia de Big Data y Business Intelligence, cuando justamente es al revés Big Data surge para poder tratar con estos datos. Por último se relaciona el concepto con riesgos de seguridad, estos conceptos no tienen vinculación alguna.

Lo que sí es evidente de las definiciones seleccionadas, es que hay conceptos que se repiten. Es por esto que se crea una definición utilizando la unión de estos conceptos.

Es por esto que el equipo de trabajo decidió crear su propia definición utilizándola unión de estos conceptos. Se puede decir que uno se encuentra con un problema de Big Data cuando:

- La estructura de los datos tienen una gran complejidad o simplemente no tienen estructura.
- El tamaño de los datos es un problema, con las tecnologías convencionales como las bases de datos relacionales.
- Se dificulta la recuperación y el procesamiento de los datos.

Se definen estas características como las dimensiones del problema de Big Data. Estos problemas surgen cuando los medios “convencionales” de almacenamiento, recuperación y procesamiento de datos no son suficientes.

Es pertinente aclarar, que uno no se encuentra ante un problema de Big Data cuando surgen problemas en todas las dimensiones, por el contrario, basta que en una de estas dimensiones sea un problema.

De los documentos seleccionados también se puede observar que los problemas surgen por alguna (o todas) de estas circunstancias:

.

- Los datos provienen de múltiples fuentes que dependen entre sí, generando así los datos no estructurados.
- Una gran cantidad de permutaciones de los datos son de utilidad.

Es por todo esto que una solución de Big Data es aquella que soluciona uno o más de los problemas en las dimensiones definidas para un problema determinado. Entonces se puede decir que existen distintas soluciones de Big Data para distintos problemas.

3.1.3 Clasificación de Herramientas de Big Data

Luego de haber definido el término Big Data se procedió a realizar un relevamiento de distintas herramientas de Big Data. Se observó que se pueden catalogar en base el tipo de trabajo que realizan. Se identificaron tres tipos de categorías:

- **Herramientas de análisis:** Son las que infieren nueva información a partir de los datos.

.

- **Herramientas de almacenamiento:** Se encargan de solucionar los problemas en cuanto al almacenamiento y recuperación de información.

- **Herramientas de procesamiento:** Se encargan de procesar los datos de manera más eficiente. Lo hacen en ambientes altamente concurrentes y distribuidos para aprovechar grandes cantidades de recursos informáticos en el manejo de grandes volúmenes de datos

A lo largo de esta sección se profundiza sobre las tres.

3.1.3.1 Almacenamiento

Como ya se mencionó en esta categoría se encuentran aquellas herramientas creadas para el almacenamiento de datos. Para este tipo de herramientas la industria se ha encargado de darle su propio nombre NoSQL.

NoSQL es la combinación de dos palabras, “No” y “SQL”. Lo que se busca es indicar que no es un sistema de gestión de bases de datos relacionales (RDBMS por su sigla en inglés).

Hoy en día el término NoSQL se utiliza para agrupar productos (base de datos) que no siguen los principios de los RDBMS.

En la Figura 1se muestra cómo se mantiene el rendimiento de las bases de datos no relacionales a medida que aumenta el volumen de datos en comparación con las bases de datos relacionales.

Las bases de datos no relacionales ponderan el rendimiento ante otros atributos de las bases de datos relacionales (como podría ser la consistencia de datos), a continuación veremos algunos aspectos teóricos para poder entender porque la ponderación del rendimiento puede no ser compatible con el modelo relacional.

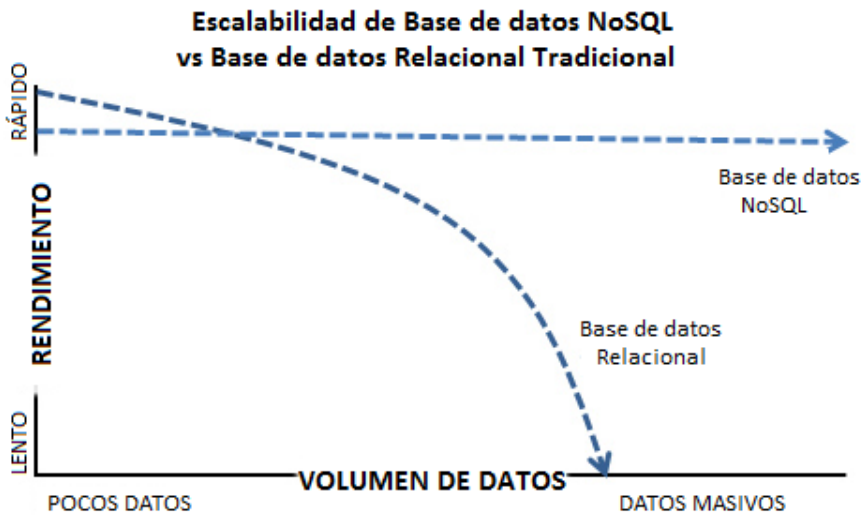

Figura 1 - Diferencias de escalabilidad entre NoSQL y RDBMS [14]

3.1.3.1.1 ACID vs BASE

Al usar el modelado de datos NoSQL se está violando alguna de las propiedades ACID de los motores de base de datos convencionales (para más información ver anexo Teorema CAP), en cambio se nos prestan las propiedades BASE descriptas a continuación. [15] [16]

- **Basically Available**: El sistema no garantiza la disponibilidad de los datos. Habrá una respuesta para cada petición pero esta respuesta puede ser un fallo al intentar obtener los datos solicitados o los datos pueden estar en un estado inconsistente o cambiando de estado.

- **Soft State**: El estado del sistema puede cambiar con el tiempo. Incluso en momentos donde el sistema no está siendo usado pueden estar realizándose cambios para alcanzar la consistencia (Eventually consistent).

- **Eventually consistent**: Se alcanza consistencia a lo largo del tiempo. La información se propaga para todas partes pero no se controla la consistencia en cada transacción antes de pasar a la siguiente. [17]

A continuación veremos los tipos de taxonomías de bases de datos NoSQL que implementan estas propiedades.

3.1.3.1.2 Tipos o Taxonomías

Existen diferentes tipos de bases de datos NoSQL que se encargan diferentes problemáticas. A continuación se realiza un análisis de los diferentes tipos (o taxonomías), en la Figura 2 se muestra una representación visual de las mismas.

- ***Documento***
  El almacenamiento se realiza en documentos, estos tienen por lo general información en algún formato estándar (xml, yaml, json, bson, pdf, etc.). Cada producto tiene alguna forma de relacionar o agrupar documentos: colecciones, tags, metadata no visible o jerarquía de directorios. Los documentos son identificados en la base de datos por una clave (clave-documento) pero no solo por la clave pueden ser buscados los documentos sino que también se provee una API o lenguaje de consulta que nos permitirá buscar documentos por el contenido de los mismos.

- ***Grafos***
  La clave de este modelo es la relación entre los datos. Se construye un grafo que conecta datos que tengan una relación entre ellos. Ejemplos de posibles datos: topologías de red, mapas de carreteras, paradas de ómnibus, etc.

- ***Clave-valor***
  Es el modelo más simple. Cada dato en la base de datos está identificado por una clave. Es un hash que permite buscar el dato en un orden uno promedio.

- ***Columna***
  Los RDBMS almacenan los datos por fila pero en este modelo se almacenan por columna. Este modelo es más eficiente ya que evita el almacenamiento de nulos, o sea si no existe el valor para esa columna no se almacena esa columna.

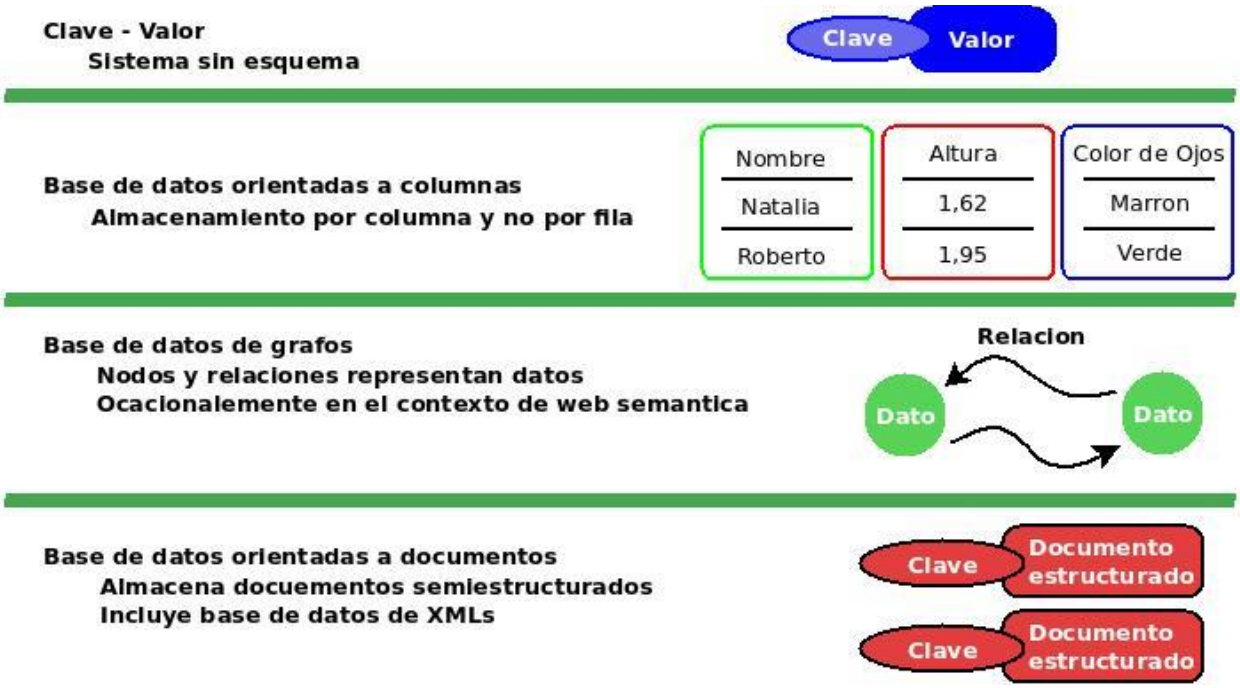

Figura 2 - Representación visual de las taxonomías [18]

3.1.3.1.3 Productos organizados por taxonomía

A continuación se muestran algunos productos clasificados por su taxonomía con algunas de sus principales características [19]. En la Tabla 1 se muestran bases de datos orientadas a grafos, en la Tabla 2 las orientadas a columnas, en la Tabla 3 se muestran las bases de datos orientadas a documentos, por último en la Tabla 4 se encuentran ejemplos de bases de datos de clave-valor.

Grafo	
Neo4J [20]	**API:** Muchos lenguajes **Consultas:** SparQL, native Java API, Ruby **Replication:** MySQL master/slave
Infinity Graph [21]	**API:** Java **Consultas:** Api de navegación de grafo, lenguaje de clasificación de predicados
Titan [22]	**API:** Java, Blueprints, Gremlin, Python, Clojure **Consultas:** Gremlin, Sparql

Tabla 1 – Bases de datos orientadas a grafos

Columna	
HBase [23]	**API:** Java **Consultas:** MapReduce **Replicación:** HDFS
Cassandra [24]	**API/Consultas:** CQL y Thrift **Replicación:** Peer to Peer
Hypertable [25]	**API:** Thrift **Consultas:** HQL y Thrift **Replicación:** HDFS

Tabla 2 - Bases de datos orientadas a columnas

Documento	
MongoDB [26]	**API:**BSON **Consultas:** MapReduce y OOP **Replicación:** Master Slave y Auto-Sharding
CouchDB [27]	**API:** Json **Consultas:** MapReduce y Javascript **Replicación:** Master
ClusterPoint [28]	**API:** XML, PHP, Java, .NET **Consultas:** xml, range y xpath **Replicación:** Multi Master Cluster

Tabla 3 - Bases de datos orientadas a Documentos

Clave-Valor	
Redis [29]	**API:** Muchos lenguajes **Replicación:** Master Slave
RiakKV [30]	**API:** Json **Consulta:** MapReduce
Berkeley DB [31]	**API:** Muchos lenguajes **Replicación:** Master Slave

Tabla 4 - Bases de datos de Clave – Valor

3.1.3.2 Consulta y análisis de datos

Otro conjunto de herramientas que se distingue son las herramientas destinadas para el análisis de datos. Es en esta categoría que colocamos a las herramientas que infieren nueva información a partir de información ya almacenada, por ejemplo, en una base de datos NoSQL.

Dentro de esta categoría entran herramientas de consulta (cómo indexadores, buscadores, etc.), herramientas de análisis (cómo herramientas de aprendizaje automático) y herramientas de trabajo con múltiples fuentes.

En la Tabla 5 se presentan algunos ejemplos de herramientas de Big Data para el análisis de datos.

Herramientas de Big Data - Análisis	
Apache Solr [32]	Motor de búsquedas en texto, también sirve como almacenamiento representando una base de datos NoSQL basada en la taxonomía de modelo de datos Key - Documento XML, también provee búsqueda de facetas de bajo costo, almacena todo en memoria y de forma distribuida.
Apache Chukwa [33]	Herramienta construida sobre Hadoop para monitorizar la producción de datos de manera distribuida, mejorando la recolección y el análisis de estos datos.
Apache Pig [34]	Plataforma para realizar análisis sobre grandes conjuntos de datos. Define un lenguaje de alto nivel para realizar programas de análisis de datos, acoplado a una infraestructura para evaluar los resultados. Consiste en un compilador del lenguaje de consultas definido por Pig (Pig Latin) que da como resultado una secuencia de programas Map/Reduce para que sean aplicados por ejemplo por Hadoop.
Apache Mahout [35]	Conjunto de librerías de aprendizaje automático diseñadas para ser escalables y robustas. Corre sobre Hadoop utilizando el paradigma map/reduce (aunque no necesariamente todos los algoritmos lo requieren). Se aplica básicamente en 3 tipos de casos de uso: recomendación basada en datos de uso, clasificación automática en base de corpus y clustering.
GraphLab [36]	Framework basado en grafos, de alta performance y distribuido principalmente usado para aprendizaje automático. Puede correr sobre el filesystem de Hadoop (HDFS). Alguno de sus módulos sirven para: análisis de grafos, clustering, filtrado colaborativo, modelos de grafos.
Google BigQuery [37]	Es un sistema pago que permite analizar grandes volúmenes de datos en la nube de Google. Ejecuta consultas tipo SQL rápidas en conjuntos de datos de varios terabytes en cuestión de segundos. Escalable y fácil de utilizar, también ofrece información detallada en tiempo real sobre los datos.
Microsoft HDInsight [38]	La solución de Microsoft al problema de big data está fuertemente relacionada con Azure, Hadoop y Mahout. Al ser la combinación de tecnologías se logra tener los beneficios de cada una de ellas.

Tabla 5 - Herramientas de Big Data para análisis

3.1.3.3 Procesamiento

Existen herramientas cuyo propósito es poder procesar conjuntos de datos que están bajo alguno de los escenarios que plantea Big Data. Una de las principales preocupaciones de estas herramientas es que el procesamiento de los datos sea llevado a cabo con altos grados de paralelización.

Se puede encontrar dos grandes grupos de herramientas dentro de esta categoría, aquellas en donde el procesamiento es llevado a cabo por procesos en lote (por ejemplo Apache Hadoop [39]), y las que se encargan de procesar los datos en tiempo real o sobre flujos continuos de información (por ejemplo Apache Storm [40]). En la Tabla 6 se puede ver algunos ejemplos de herramientas de Big Data utilizadas para el procesamiento de datos.

Herramientas de Big Data - Procesamiento	
Apache Hadoop [39]	Framework para mejorar el procesamiento distribuido de gran cantidad de datos de forma escalable y con alta disponibilidad.
Apache Sqoop [41]	Herramienta diseñada para migrar una gran cantidad de datos producidos por el procesamiento de Hadoop de manera eficiente a un manejador de datos estructurados como puede ser un RDBMS.
Apache Spark [42]	Herramienta para procesar las fuentes de datos de Hadoop. Puede correr sobre el administrador de clúster YARN de Hadoop 2
Apache Storm [40]	Es un sistema libre y open source de computación distribuida en tiempo real. Este sistema se especializa en realizar de forma sencilla el procesamiento de flujos de datos no acotados.
Microsoft HDInsight [38]	La solución de Microsoft al problema de big data está fuertemente relacionada con Azure, Hadoop y Mahout. Al ser la combinación de tecnologías se logra tener los beneficios de cada una de ellas.
Apache Oozie [43]	Es un planificador de workflows para manejar múltiples flujos Hadoop. Simplifica los flujos de trabajo y la coordinación entre cada uno de los procesos.
Apache Flume [44]	Es un servicio para recolectar y mover grandes cantidades de datos. Su tarea principal es dirigir los datos de una fuente hacia alguna otra localidad.

Tabla 6 - Herramientas de Big Data para procesamiento

3.1.3.3.1 Map Reduce

Uno de los paradigmas que ha sobresalido en cuanto a procesamiento distribuido es MapReduce, el cual sirve para procesar grandes conjuntos de datos en un clúster de

ordenadores. El primer framework basado en este paradigma fue patentado por Google [45], pero existen implementaciones de código abierto, siendo Hadoop una de las que destaca.

En Map Reduce, el usuario especifica una función map que procesa un par clave-valor y genera un conjunto intermedio de clave-valor. La función reduce hacer el merge de los valores intermedios que comparten la misma clave. Programas escritos de forma funcional son automáticamente paralizados y ejecutados en un gran clúster de ordenadores. Este comportamiento se puede ver en la Figura 3

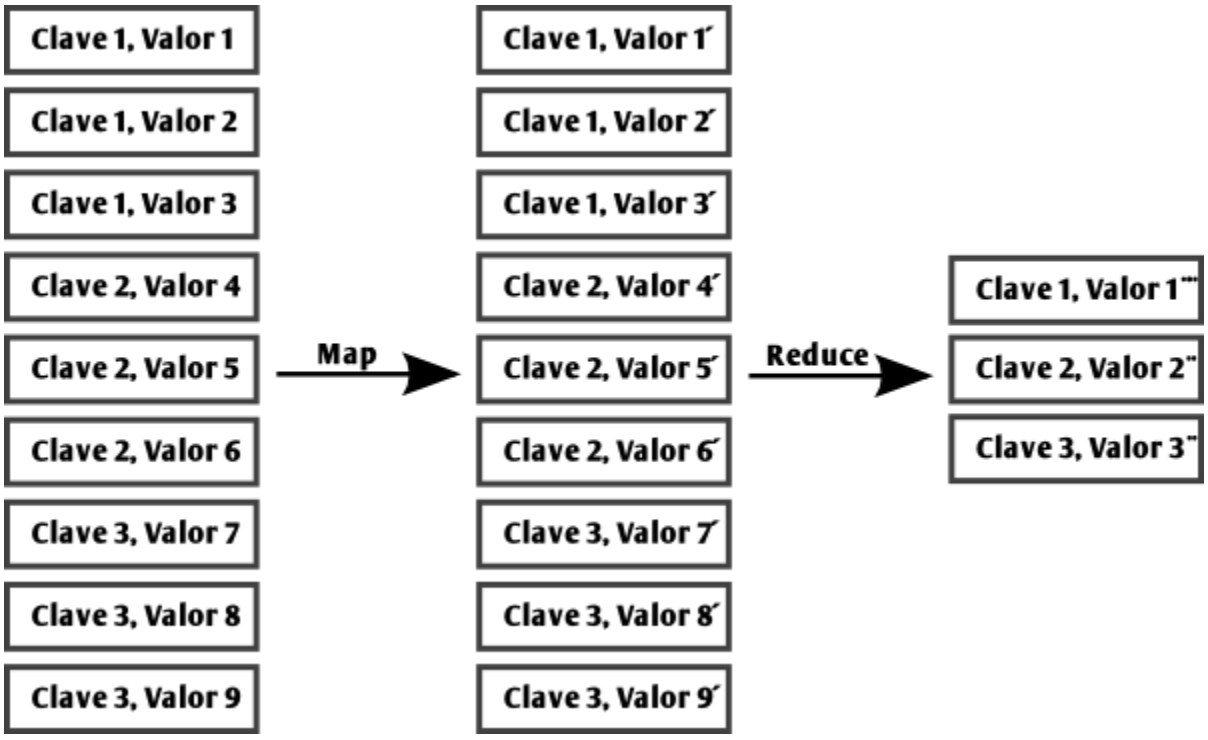

Figura 3 - Proceso Map Reduce

3.2 Workflows

Los workflows como concepto, fueron definidos por primera vez en el mundo empresarial por la Workflow Management Coalition [46] cómo:

> *"The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules."*

La automatización de un proceso de negocio, en su totalidad o en parte, durante el cual los documentos, información o tareas pasan de un participante a otro para realizar una acción, de acuerdo a un conjunto de reglas." [47]

Existen sistemas que permiten crear, ejecutar y administrar workflows. Estos sistemas son conocidos como sistemas de administración de workflows (WfMS por sus siglas en inglés).

Es entonces un workflow es "una forma de modelado y especificación con primitivas, para optimizar, especificar y automatizar la ejecución de procesos, organizando, controlando y monitoreando las tareas" [48].

3.2.1 Workflows empresariales

Los workflows empresariales han estado en la industria ya por varios años. Su objetivo es optimizar los procesos realizados por personas o computadoras desde un punto de vista administrativo. Estos últimos ocupan roles y se encargan de manipular recursos y especialmente la coordinación y el orden parcial de actividades. Hoy en día existen cientos de sistemas de workflows. [48]

Los workflows empresariales se caracterizan por tener un modelo de computación orientado a flujo de control, refiriéndose a modelo de computación a la forma en la que ejecutan las tareas del workflow. En el caso de flujo de control determina que la ejecución entre las tareas es de forma serial, si una tarea precede a otra. [48]

3.2.2 Workflows científicos

Los sistemas de workflows científicos (WfMSs) son sistemas desarrollados para la automatización de experimentos científicos que necesitan lidiar con grandes cantidades de datos. Su principal objetivo es la reutilización e integración de herramientas y funciones específicas de dominio a través de una interfaz gráfica. [49]

Los workflows científicos son ampliamente reconocidos como un paradigma útil para describir, administrar y compartir los análisis científicos complejos [49]. Son el método usado a menudo para la experimentación *in silico*.

Dentro de un workflow pueden realizarse varios tipos de tareas: servicios web remotos, scripts y sub-workflows (workflows completos utilizados como subrutinas en otro de mayor tamaño). Cada una de estas tareas sólo es responsables de un pequeño fragmento de funcionalidad. Estas tareas se encadenan en un grafo dirigido acíclico con el fin de obtener un flujo de trabajo que puede realizar una tarea útil

3.2.3 Comparación Workflows Científicos vs Workflows Empresariales

Afortunadamente los workflows científicos tienen muchas cosas similares a los workflows empresariales, por esta razón parece interesante plantear los requerimientos de un sistema de workflows científico desde las diferencias con los workflows empresariales.

Basados en los trabajos “Business versus Scientific Workflows: A Comparative Study” [48] y “Pattern-Based Evaluation of Scientific Workflow Management Systems” [49] se busca definir cuáles son las principales diferencias entre los dos modelos. A continuación se plantea punto por punto las diferencias entre ambos modelos y se toma cada diferencia como un requerimiento a cumplir en los sistemas de workflows científicos.

3.2.3.1 Objetivos de workflows científicos vs empresariales

Antes de plantear las diferencias, se deben definir objetivos claros de los sistemas de administración de workflows (WfMS). Estos son:

- Modelar y especificar procesos con primitivas previamente diseñadas.
- Reingeniería de los procesos diseñados para su verificación y optimización.
- Automatizar la ejecución de los procesos a través del control, la organización y el monitoreo de las tareas.

La principales diferencias de los sistemas de workflows empresariales y científicos vienen dadas por los objetivos, en los workflows empresariales se intenta optimizar los procesos donde los actores son personas o máquinas desde un punto de vista administrativo, donde algún orden parcial de ejecución de las tareas es requerido. Por otro lado cuando los workflows se mueven al mundo científico, estos son ejecutados por un solo científico o un número reducido de ellos. Además pueden ejecutar todas las tareas concurrentemente ya que los workflows científicos son orientados a un flujo de datos. [49]

3.2.3.2 Orientados a flujos de datos vs flujos de control

El modelo de computación orientado a flujo de datos refiere a que un nodo de un workflow científico empieza a ejecutar cuando tiene datos disponibles producidos por un nodo anterior. En los workflows empresariales el orden determina que una tarea B no puede ejecutar hasta que una tarea A termine si hay un orden de precedencia definido. Esta forma de ejecución de los workflows científicos es la que los define como orientados a flujos de datos.

En la Figura 4 se ve un workflow definido para ambos paradigmas. En la segunda sección de la imagen llamada Interpretación del Diseñador (Designer Interpretation) se puede ver la diferencia. Para el workflow científico, la precedencia entre los nodos F y E identifica que una

tarea F recibe los datos de una tarea E. Para el workflow empresarial, la precedencia indica la tarea E debe terminar de ejecutar antes de empezar a ejecutar la tarea F.

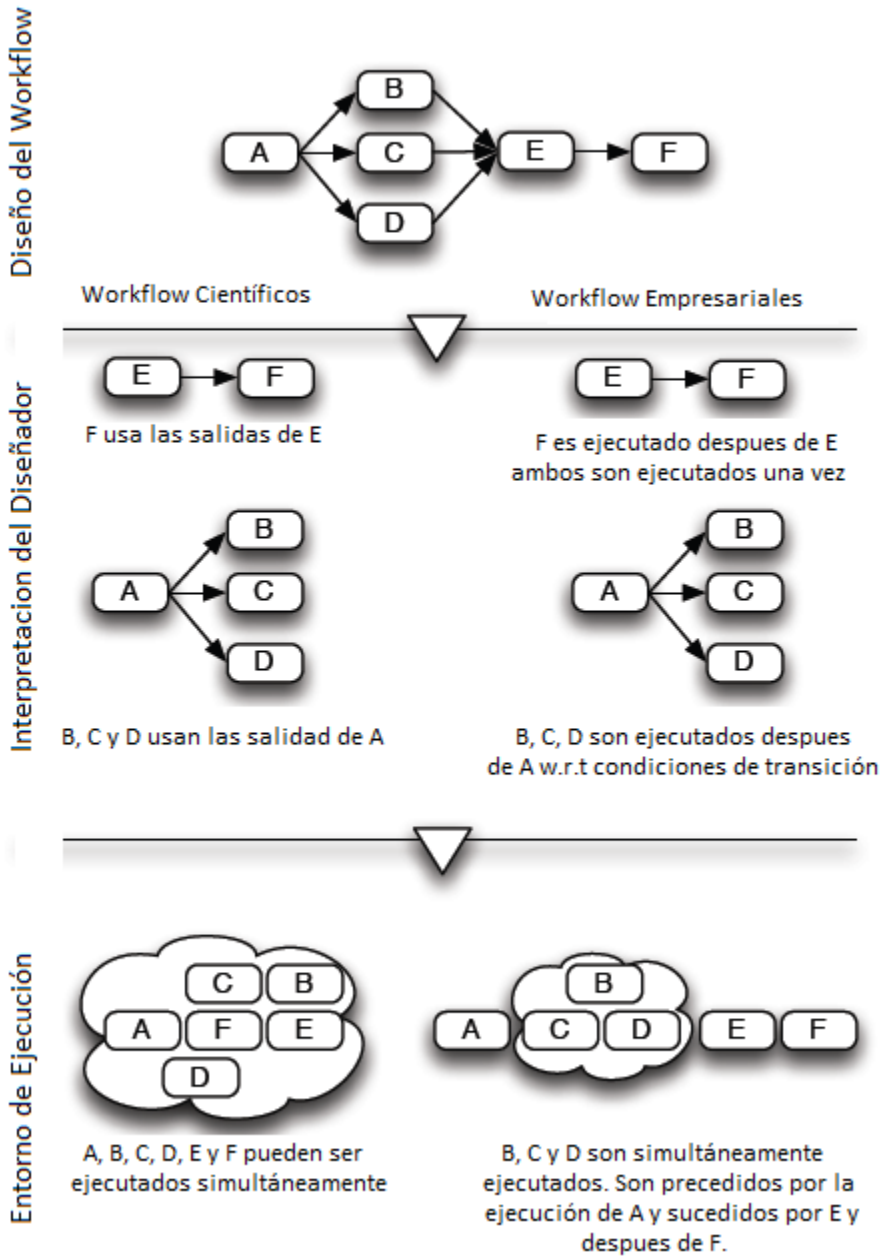

Figura 4 - Comparación de comportamientos de workflows [48]

Por esto se dice que un requerimiento de los workflows científicos es que son **orientados a flujos de datos y los workflows empresariales orientados a flujos de control.**

3.2.3.3 Alta concurrencia

En los workflows empresariales es importante el orden de ejecución de las tareas, dado que la ejecución de una depende de sus predecesoras. El paradigma de workflows científicos es totalmente diferente. Las tareas tienen 100 % de concurrencia mientras el flujo de datos pasa por ellas. Esto se muestra en la sección Entorno de Ejecución (*Execution Environment)* en la Figura 4.

3.2.3.4 Patrones multi instancia

En el artículo "Business versus Scientific Workflows: A Comparative Study" [48] se realiza un estudio de la implementación de los patrones de ejecución para los cuales se requieren múltiples instancias, tanto para una tarea dentro de un nodo de un workflow como para el workflow entero.

Los patrones multi instancia describen situaciones en las cuales múltiples hilos de ejecución están activos dentro de un proceso asociado a una tarea. Los sistemas de workflows científicos se dicen ser más apropiados para implementarlos mientras que en los sistemas de workflows empresariales no es tan trivial [48].

Para que estos patrones puedan llevarse a cabo, los workflow deben proveer de alguna primitiva "bundle" que organice el flujo luego de la ejecución múltiple. En un ambiente paralelo y altamente distribuido la implementación de este control podría llegar a no ser trivial. En el artículo [48] pueden verse múltiples patrones con distintas realidades.

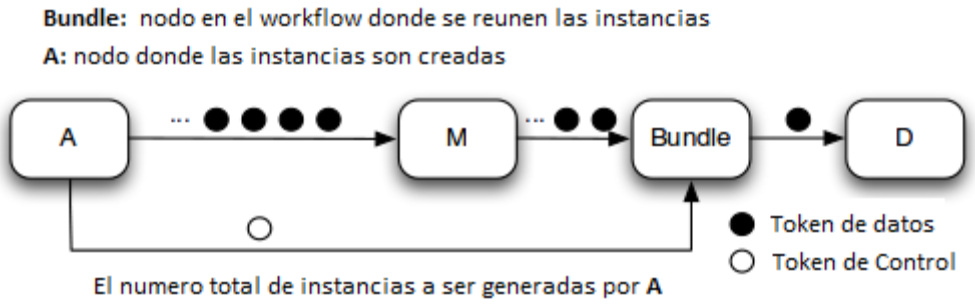

Figura 5 - Patrón multi instancia workflow científico [48]

En la Figura 5 podemos ver como un patrón multi instancia típico es implementado en un workflow científico, la tarea A genera el flujo de datos que van a la tarea M, la cual se ejecuta múltiples veces en paralelo, luego la directiva bundle a través de alguno token de control ordena el flujo proveniente de las múltiples instancias de M y los ordena para poder continuar ejecutando en D.

En la Figura 6 se implementan tres maneras diferentes de este comportamiento. Las partes (a) y (b) están construidas con primitivas, las que algunos sistemas de workflows empresariales proveen, en la parte (c) se crean múltiples ramas de ejecución con el número de instancias que debería tener M.

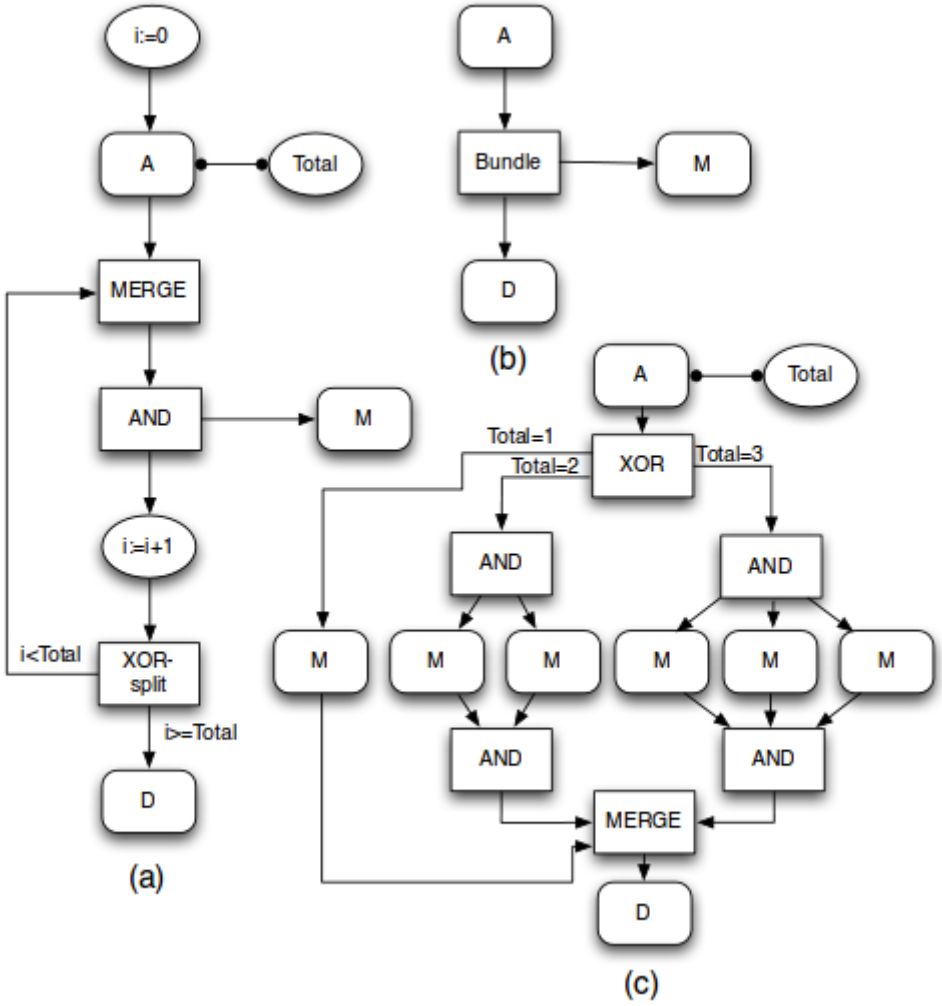

Figura 6 - Comportamiento típico de un workflow empresarial [48]

Tanto en el workflow (a) cómo el (b) la instancia A es la que sabe cuántas instancias del nodo M deben ser creadas. En la figura (a) se coloca un contador que es incrementado hasta que se alcanza el número de instancias de M necesarias antes de activar el nodo D. En el workflow (c) ya se sabe antes de ejecutar cuantas instancias del nodo M se precisan y simplemente se agrega esta cantidad de nodos de M, se utiliza la primitiva MERGE para ordenar los datos. En (b) se asume que existe una primitiva Bundle que puede crear tantas instancias de una actividad dado un número, es decir en tiempo de ejecución esta primitiva Bundle recibe un número y crea esa cantidad de instancias de la actividad [48].

3.2.3.5 Estructuras iterativas

Las estructuras iterativas en los workflows científicos se comportan diferentes a como lo hacen los workflows empresariales, donde las iteraciones se construyen con primitivas en el diseño del workflow. En los workflows científicos las iteraciones son implícitas ya que el flujo de datos no puede volver “hacia atrás”. Esto está dado porque por definición, los workflows científicos son grafos dirigidos acíclicos. De manera similar pasa con la recursión, no se puede colocar la salida de un nodo al mismo nodo.

3.2.4 Abstracción de la computación

Este punto refiere a la usabilidad del software y no se presentada en contraposición a los workflows empresariales.

Es necesario que los sistemas transmitan lo menos posible en cuanto a la computación de los recursos a la interfaz del usuario, ya que el científico, biólogo, etc.; no debería ser un especialista en computación, solo debería tener conocimiento en la construcción de flujos de trabajos.

Este punto se recalca ya que en este proyecto se quiere aplicar soluciones de Big Data en los Workflows científicos, impactando lo menos posible en el usuario.

3.2.5 Relevamiento de herramientas de workflows científicos

A continuación veremos algunos aspectos de las dos soluciones de workflows científicos (de ahora en adelante SWfMS (Scientific Workflow Management Systems) open-source más populares.

3.2.5.1 Kepler

Este SWfMS tiene el paradigma de modelado “orientado a actores”, los nodos de ejecución en un workflow en Kepler [50] son llamados “actores” los cuales se comunican entre ellos a través de “ports”, estos básicamente definen los parámetros de entrada y salida de un actor, y los actores se conectan a través de “canales”, estos canales representan un único flujo de datos.

Además de esto también tiene el concepto de “director”, para un workflow en Kepler el director determina cómo debe ser ejecutado, dando la opción de ejecutar distintos modelos de computación (como orientado a flujos o de forma imperativa, entre otros) e incluso combinarlos, haciendo de Kepler un SWfMS muy versátil en este aspecto.

En la Figura 7 se ve un workflow definido en Kepler

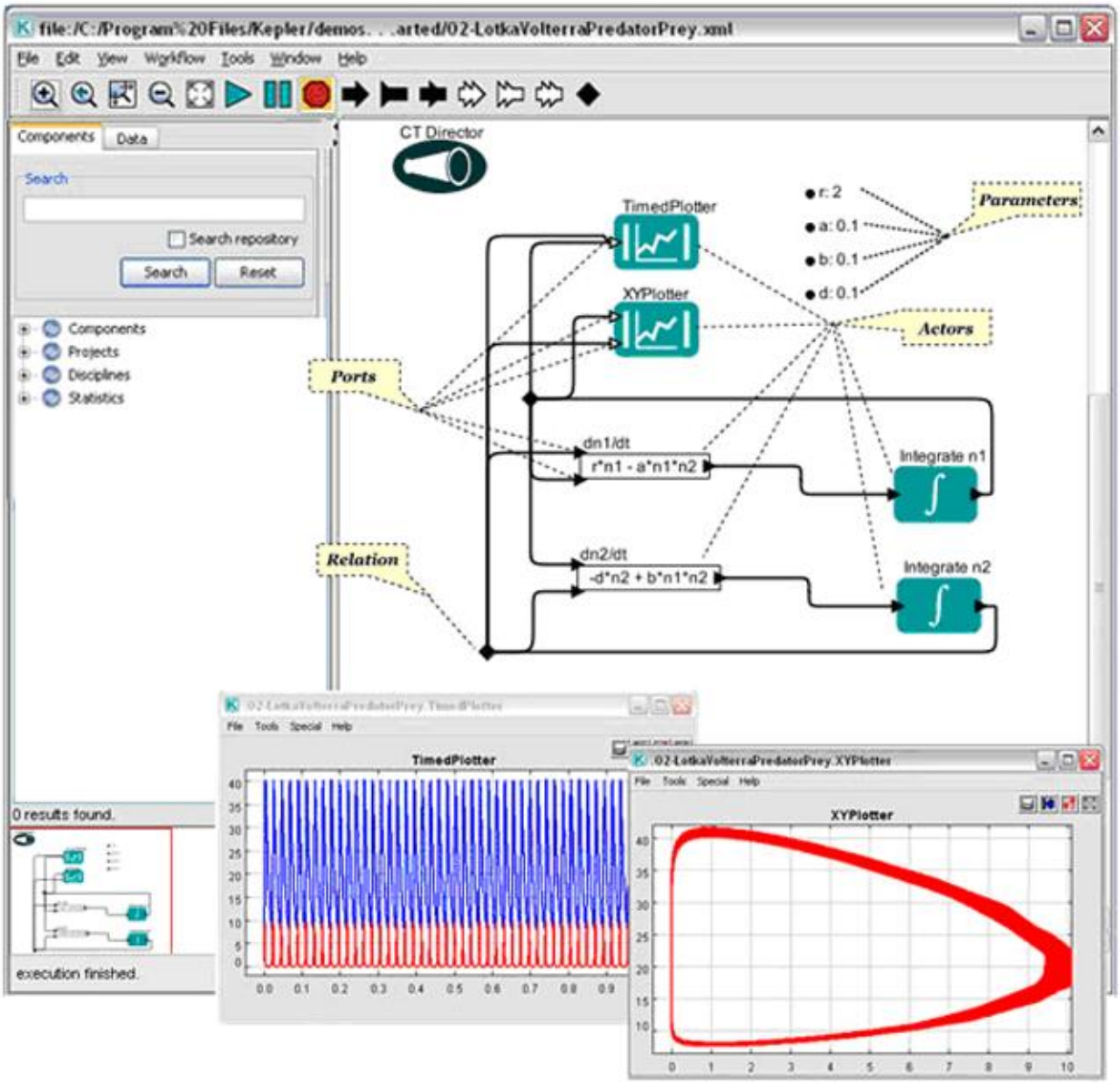

Figura 7 - Interfaz gráfica de Kepler

En cuanto a las características que previamente se definieron, Kepler consta de un modelo de computación orientado a flujos. En cuanto a alta concurrencia, podría llegar a ejecutar en múltiples paradigmas de ejecución e incluso en múltiples entornos de ejecución como se ven por el trabajo realizado en [51]. La implementación de los patrones multi instancia es soportado por Kepler, aunque la ejecución del mismo dependerá del director en uso. Los patrones de estructuras iterativas están dados implícitamente. En cuanto a modelos de computación, este es el fuerte de Kepler ya que el concepto de director lo hace extremadamente versátil. En cuanto a interfaz de usuario no es muy diferente de las interfaces tradicionales de diseño de workflows, pero en algunos casos requiere que se sepa demasiado el modo de ejecución del workflow, requiriendo habilidades en computación.

3.2.5.2 Taverna

Taverna [52] es un SWfMS open-source desarrollado por la Universidad de Manchester, sus principales fortalezas son que cuenta con una gran cantidad de servicios web externos organizados que pueden ser usados en los workflows y además utiliza Scufl, un lenguaje gráfico para la especificación de los workflows.

El modelado en Taverna viene dado por “procesadores”, estos son los nodos del workflow que ejecutan alguna acción. Cada procesador tiene definidos “ports” de entrada y de salida, estos se conectan con otros procesadores a través de “enlaces de datos”. Además también cuenta con “enlaces de coordinación” que son enlaces de control por los que no van datos, esto viene dado porque Taverna solo cuenta con un modelo de computación orientado a flujos, por lo que necesita algún mecanismo para definir la coordinación entre los procesadores.

En la Figura 8 se puede observar la definición de un workflow en Taverna

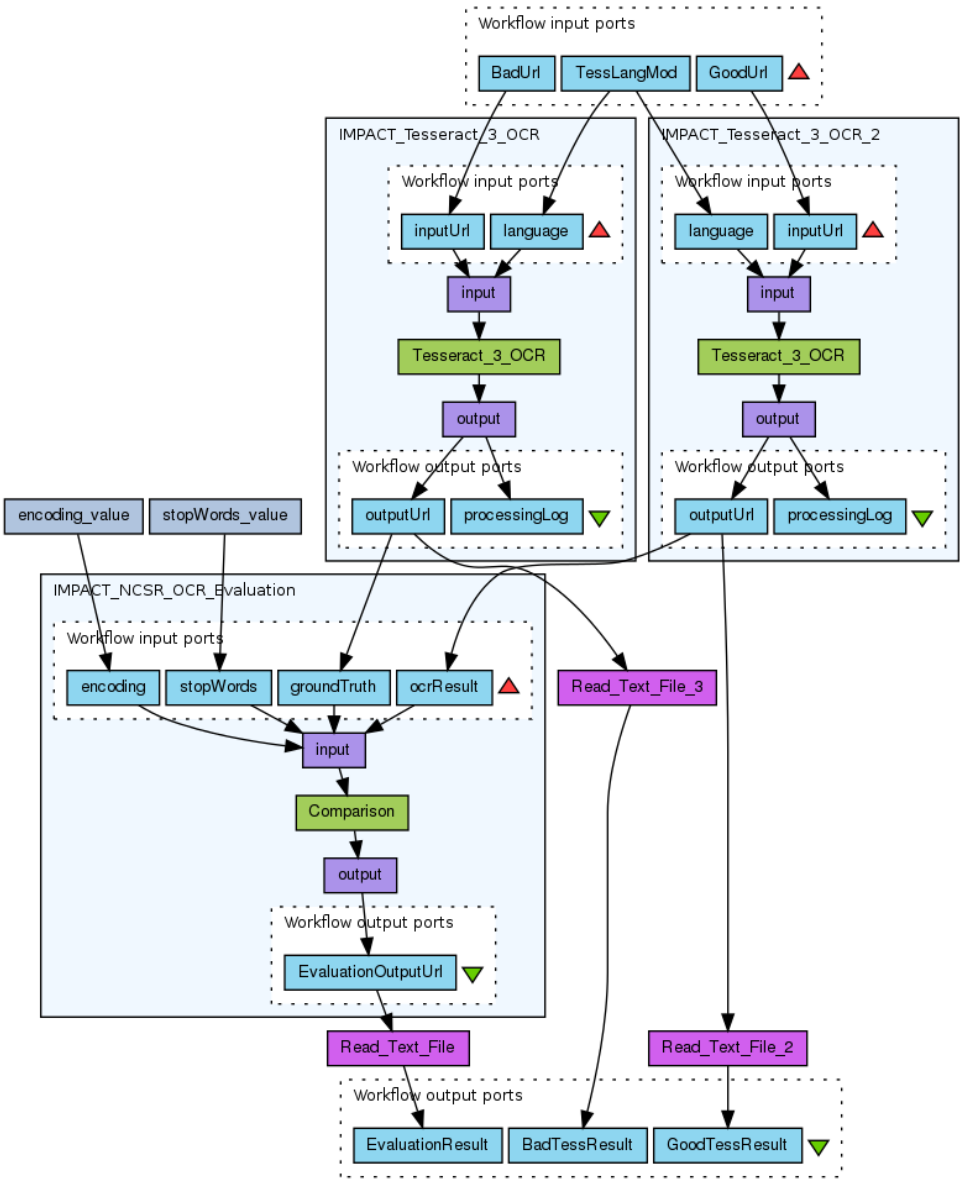

Figura 8 - Workflow definido en Taverna

En cuanto a los requerimientos que previamente se definieron Taverna solo cuenta con un modelo de computación orientado a flujos, puede ejecutar los procesadores en paralelo pero no de manera distribuida, para ello debe utilizarse un plugin [53] o similar. Puede implementar patrones multi instancia y estructuras iterativas, estas últimas utilizando “enlaces de control”. En cuanto a modelos de computación solo cuenta con un modelo orientado a flujo de datos, siendo este aspecto el más débil en contraposición con Kepler. De todas maneras Taverna abstrae mucho mejor la computación para el científico usuario ya que no debe especificar modelos de computación o cosas similares con tanto conocimiento.

En la Tabla 7 se puede ver un resumen de la comparación.

	Kepler	Taverna
Paradigma	Orientado a actores	Red de procesadores
Modelo de ejecución	Varios	Orientado a flujo de datos
Concurrencia	Si	Si
Patrón multi instancia	Si	Si
Nivel de abstracción de computación	Medio	Alto

Tabla 7 - Comparación de Taverna y Kepler

3.3 Herramientas de Big Data Candidatas

En esta sección se ven dos herramientas de Big Data que el grupo considera pueden usarse para mejorar el rendimiento de un workflow científico, Apache Hadoop y Apache Storm.

3.3.1 ¿Qué es Hadoop?

En la propia página de Hadoop [39] podemos encontrar esta definición:

> *"The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models."*
>
> *La librería Apache Hadoop es un framework que permite el procesamiento distribuido de grandes volúmenes de conjuntos de datos a través de clúster de computadoras utilizando un modelo de programación simple.*

Aunque en primeras versiones de Hadoop el framework era básicamente una implementación del algoritmo MapReduce (versiones 1.*), las versiones más modernas (versiones 2.*) implementan mucho más que simplemente el algoritmo MapReduce, a continuación vemos los componentes actuales de esta plataforma.

- **Hadoop Commons:** Este es el módulo básico que le da soporte al resto de los módulos y es el corazón de Apache Hadoop.
- **Hadoop Distributed File System (HDFS):** Esta es la implementación de un sistema de archivos distribuidos que puede ser usado por las aplicaciones desarrolladas con

Hadoop, para poder así tener una visión unificada de un sistema de archivos para las tareas distribuidas.

- **Hadoop YARN:** Este es el módulo encargado de manejar y organizar la ejecución de las tareas en el clúster Hadoop, además de la gestión del mismo.

- **Hadoop MapReduce:** Un sistema basado en YARN para la ejecución de tareas distribuidas en forma concurrente, para la ejecución de grandes conjuntos de datos. Este módulo implementa el algoritmo MapReduce, ya visto en el informe.

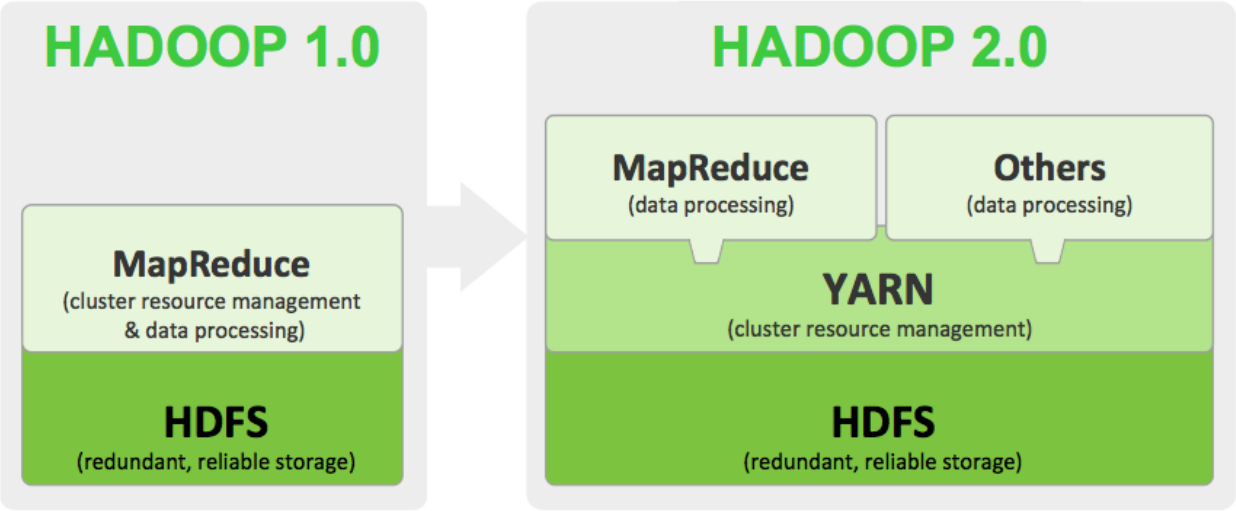

Figura 9 - Comparación de la versiones de Hadoop

En la Figura 9 se muestra la arquitectura de Hadoop comparando la versión uno con la dos.

Hadoop se define a sí mismo como un procesador de consultas en batch sobre un gran conjunto de datos, ya que las tareas concurrentes, se ejecutan sobre un conjunto de datos fijos.

En la sección de Big Data y Workflows Científicos se puede leer, como en todos los casos nombrados, se incluyen los conceptos de DataSet y MapReduce, esto es porque muchos de los artículos en los que se ha intentado paralelizar y distribuir el trabajo en los SWfMS, ha incluido integraciones o implementaciones con Hadoop.

3.3.2 ¿Qué es Storm?

En el sitio de Storm [40] se puede encontrar la siguiente definición.

> *"Storm is a distributed real time computation system. Similar to how Hadoop provides a set of general primitives for doing batch processing, Storm provides a set of general primitives for doing real time computation"*

> *Storm es un sistema de computación distribuida en tiempo real. De la misma manera que Hadoop provee un conjunto de primitivas generales para el procesamiento en batch, Storm provee un conjunto de primitivas generales para la computación en tiempo real.*

De esta definición caben destacar dos cosas, la primera es notar cómo Storm se define a partir de Hadoop. Esto es un factor común de todos los frameworks de computación distribuida, ya que Hadoop es el framework de referencia y más conocido.

Por otra parte cabe decir, que cuando en la definición se refiere a computación en tiempo real, se refiere a cómo el procesamiento llega al framework y se realiza. Ya que Storm comenzará a procesar los datos a medida que vayan llegando.

3.3.2.1 Características de Storm

En la página del framework refiere que se puede implementar el mismo comportamiento que tiene Storm, manualmente configurando distintas colas y programas *workers*. Estos programas tomarán los datos de las colas, los procesarán y los colocaran en otra cola para continuar el procesamiento. Este enfoque, aunque posible, tiene sus limitaciones según la documentación de Storm [54]:

- **Tedioso:** Se perdería la mayoría del tiempo de desarrollo configurando las colas de mensajes y los workers en vez de la propia logia del procesamiento.

- **Inestable:** Generalmente tiene poca tolerancia a fallas o es muy difícil de lograr en gran medida, haciendo que sea trabajo de desarrollo, el monitorizar las colas y mantenerlas activas.

- **Dificultad al escalar:** Cuando el trabajo se vuelve demasiado para una cola de mensajes, debemos partir los datos para que vayan a réplicas de los workers. Esto implica que los workers sepan a qué otro lugar deben enviar los mensajes e incluso crear nuevas colas, lo cual puede ser tedioso.

Al mismo tiempo la documentación de Storm, define un conjunto clave de propiedades que cumple Storm.

- **Cubre un gran conjunto de casos de uso:** Storm puede ser utilizado en múltiples casos de uso, el framework puede ser usado para procesamiento de flujos de datos, computación continua, distribuir RPCs, entre otros. Esto se debe al reducido número de primitivas de Storm, que aplican a múltiples casos de uso por su simpleza y generalidad.

- **Escalabilidad:** Storm puede escalar fácilmente a una cantidad masiva de mensajes por segundo. Lo único que se debe hacer es agregar máquinas y aumentar el paralelismo de las topologías (veremos más adelante que es una topología) a través de su configuración. Además de esto Storms utilizar ZooKeeper [55] para la coordinación en el clúster, lo cual ayuda a escalar a clústeres de gran tamaño.

- **No pérdida de datos:** Este es uno de los atributos diferenciales de Storm con otras plataformas similares, Storm garantiza el procesamiento de cada mensaje que ingresa en una topología, ya que se considera que un framework de procesamiento en tiempo real con pérdida de datos, no tiene muchos casos de uso.

- **Extremadamente robusto:** En este punto refiere a la dificultad de mantener un clúster Hadoop y como, por el contrario, es muy fácil mantener un clúster Storm.

- **Tolerancia a fallas:** En caso de fallas durante la ejecución, Storm re-asignará la tarea a otro equipo del clúster para que continúe ejecutando la tarea. Las tareas en Storm ejecutar infinitamente, o hasta que el usuario las detenga.

- **No depende del lenguaje:** En este punto manifiesta que una plataforma de procesamiento en tiempo real, no debería estar limitada a una sola plataforma, por esa razón los componentes Storm pueden ser definidos en múltiples lenguajes.

3.4 Big Data y Workflows Científicos

A lo largo de un SWfMS podemos ver distintas unidades de trabajo que podrían ser distribuidas. Podríamos distribuir workflows enteros, los nodos de un workflow o construir los nodos de forma distribuible [56] y [57].

3.4.1 Paralelización de workflows Enteros

En esta posibilidad nos planteamos distribuir los flujos completos sobre distintas unidades de procesamiento. La idea es buscar la forma de conseguir que en varios lugares se realice el procesamiento de todo un flujo y no que en varios lugares se realice parcialmente el trabajo.

En este tipo de paralelización se distinguen dos casos:

1. El workflow se encarga de dividir los datos en múltiples secciones, cada juego de datos es ejecutado por el mismo flujo en diferentes unidades de procesamiento y al final se unen los datos para presentar el resultado. A continuación se presenta en la Figura 10 cómo se distribuirán los datos. Las UPs representan las unidades de procesamientos, en cada uno de ellos se coloca una copia del workflow y luego se divide el set de datos (1) en múltiples data sets para ejecutar en cada copia del workflow (2), esto nos da un

data result para cada data set (3) que luego deben ser unidos para presentar el resultado final (4). Para lograr esto los datos deben cumplir la precondición de que pueden dividirse en distintos conjuntos sin alterar el resultado final.

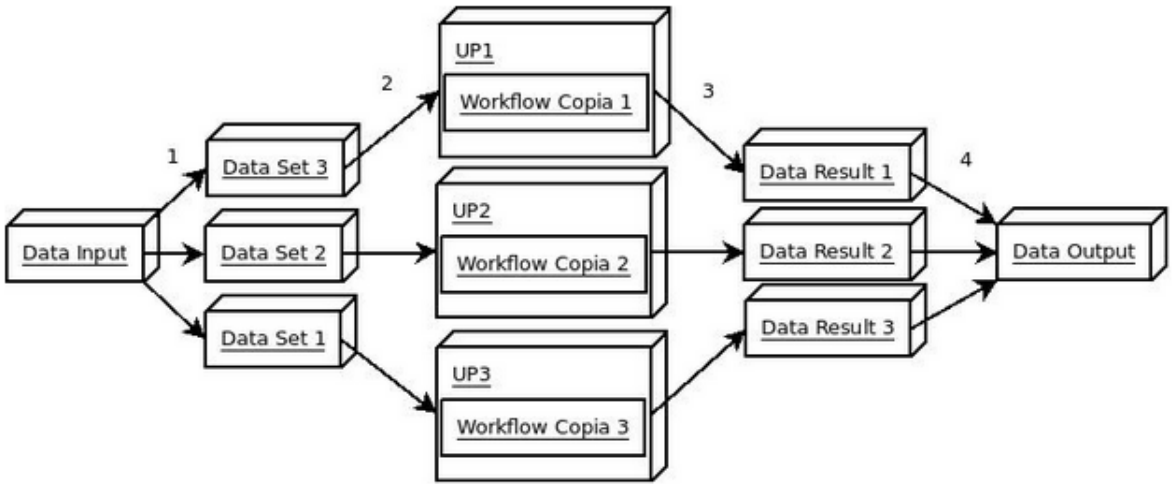

Figura 10 - El workflow parte sus datos de entrada y asigna

2. El workflow es distribuido para poder atender llamados concurrentes desde varios usuarios. Cada llamado es tratado de forma independiente. En la Figura 11 se muestran tres data sets independientes asignados a un nodo con una copia de workflow para que ejecute (1), luego cada UP devuelve un data result que se corresponde con un único data set (2).

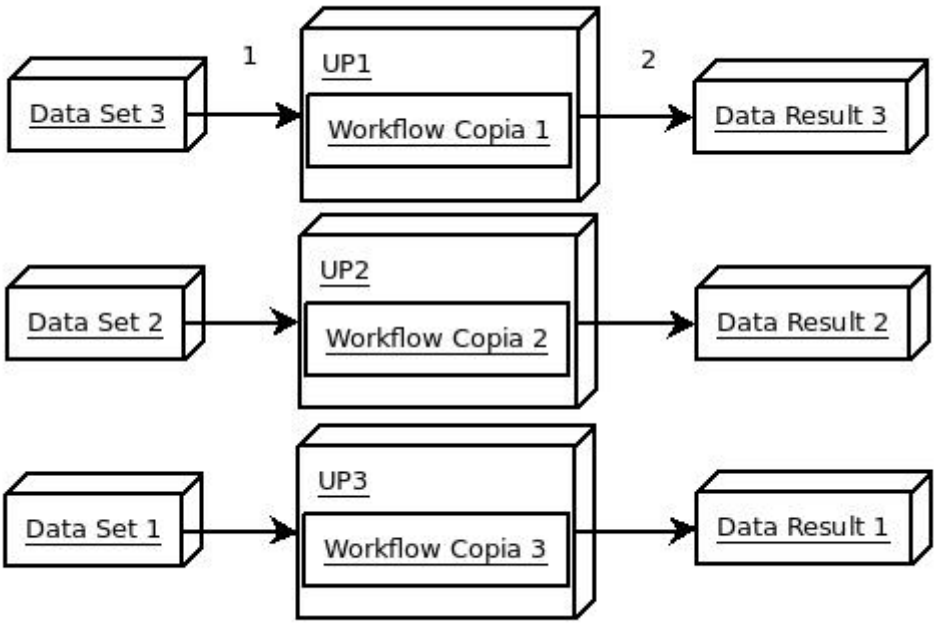

Figura 11 - Cada llamado a ejecutar provee un conjunto de datos y se ejecuta por separado

3.4.2 Paralelización de Nodos

Dentro de un SWfMS se definen distintos nodos que se encargan de realizar las tareas definidas por el usuario. Dado que algunos nodos pueden ser multi-instanciables, es de interés construir mecanismos que paralelicen estos nodos de forma transparente al usuario.

A continuación se muestra como un workflow puede ser distribuido por nodos utilizando Hadoop. En la Figura 12 cada nodo es correspondido por un job MapReduce en Hadoop, como muestran las referencias en rojo entre uno de los nodos del workflow y el servidor Job Tracker de Hadoop. El Hadoop JobTracker tomaría los nodos del workflow y los distribuirá en distintas unidades de procesamiento para que ejecuten, como se muestra en las flechas en rojo entre el Job Tracker y las distintas unidades de procesamiento del clúster. Además estos nodos tendrán referencias a HDFS para poder utilizar el sistema de archivos distribuido.

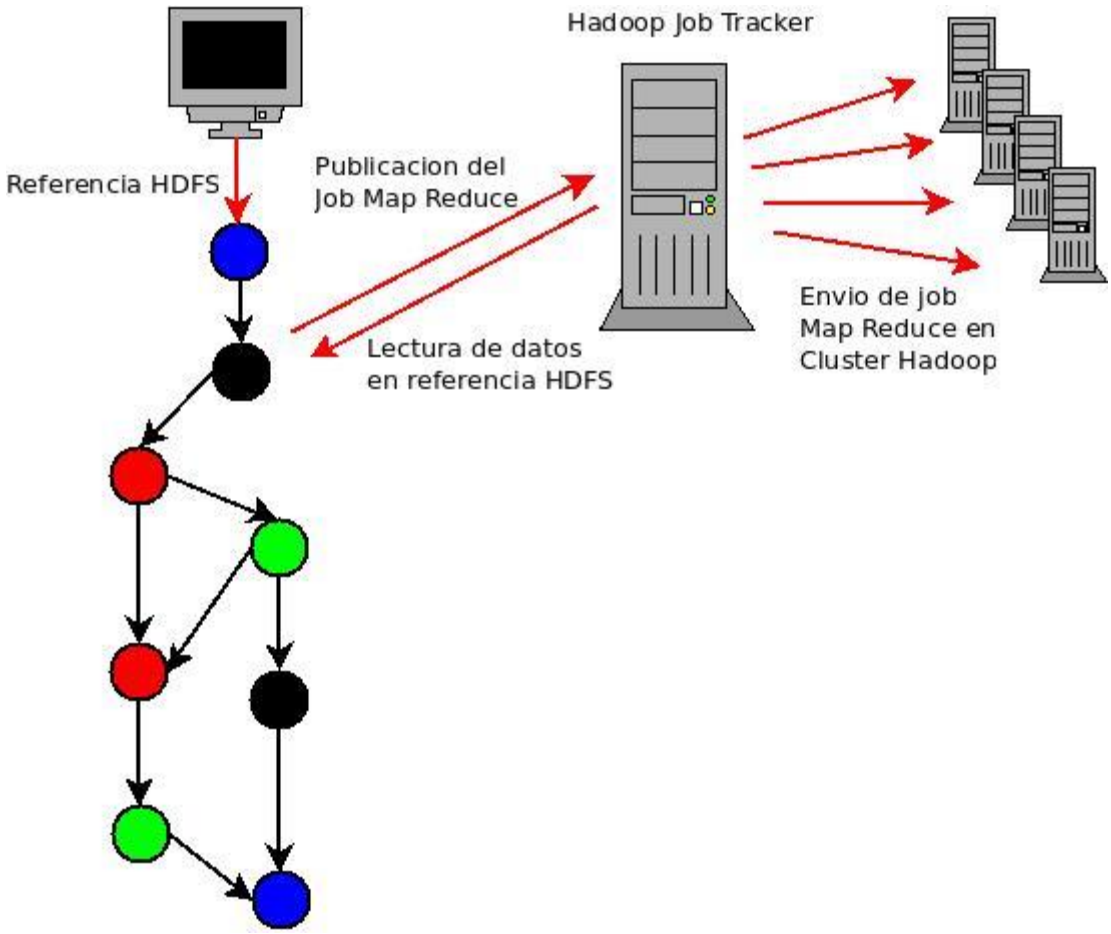

Figura 12 -Cada nodo se mapea con un job MapReduce en el clúster [56]

3.4.3 Nodos paralelizables

Dado que un usuario puede tener conocimientos para escribir procesos que trabajen de forma distribuida. Los SWfMS deberían aceptar nodos construidos en base a alguna tecnología en lo referente a Big Data, que esté preparada para afrontar este requerimiento.

El la Figura 13 tenemos una imagen tomada de [58] donde se implementa este concepto. En la imagen (a) se crea un workflow con un actor (Kepler es orientado a actores, pero perfectamente podría ser un procesador Taverna), que implementa un MapReduce determinado, esto es un sub workflow que utiliza actores MapReduce más generales pero que requieren un mejor entendimiento del algoritmo (imágenes b y c).

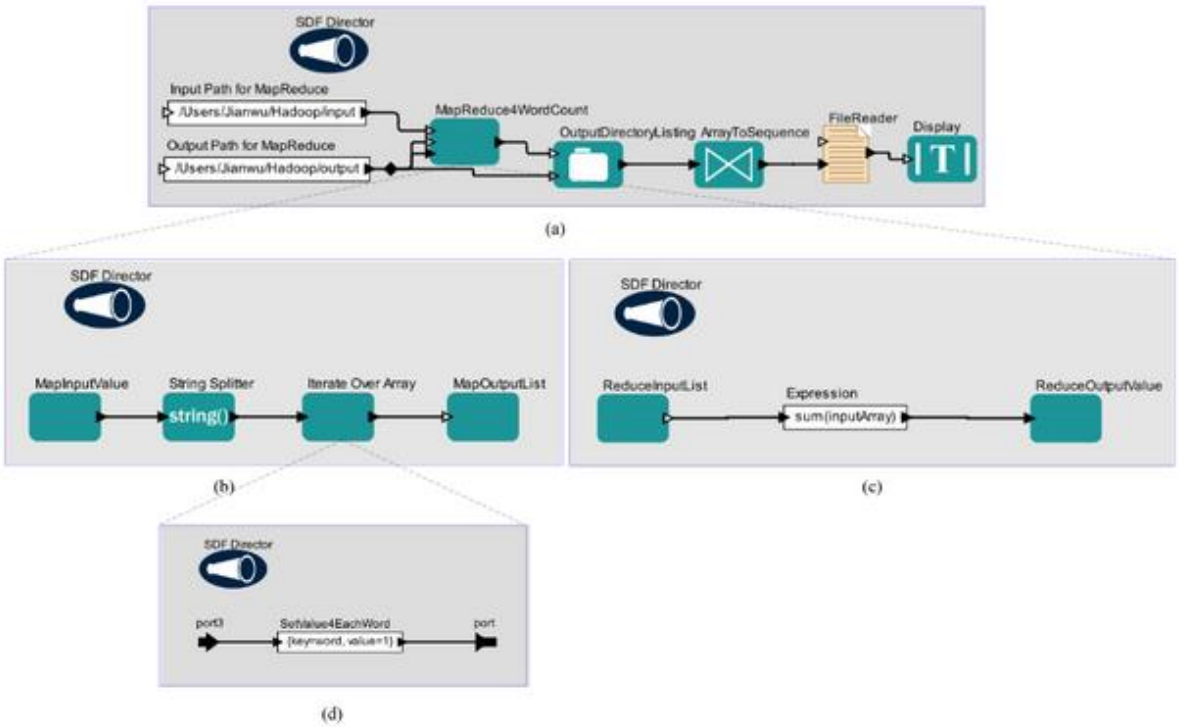

Figura 13 - Nodos paralelizables en Kepler

4 Análisis

En este capítulo se estudian las herramientas Big Data candidatas para seleccionar la que mejor se adapte a los requerimientos de un SWfMS. Además se construye una lista de los requerimientos del proyecto y el alcance del mismo.

4.1 Hadoop vs Storm

Hadoop y Storm fueron las herramientas que el equipo de trabajo consideró utilizar en el proyecto. En la sección se evalúa los principales requerimientos de un SWfMS y cómo pueden ser implementados con cada una de las herramientas. El objetivo de esta sección es ver cuál de las herramientas es más apta.

4.1.1 Workflows Científicos con Hadoop

En la sección Comparación Workflows Científicos vs Workflows Empresariales se explica que los workflows científicos son orientados a flujos de datos en vez de la precedencia y ordenamiento de tareas. Además se menciona que deben soportar una alta concurrencia de los distintos procesos que ejecutan.

4.1.1.1 Flujo de datos en Hadoop

Como se ha visto en las secciones de Big Data y Workflows Científicos, se plantean distintas maneras de paralelizar los workflows. Todos los modelos de paralelización descritos en esa sección paralelizan de alguna manera el trabajo en los workflows y todos pueden ser implementados con Hadoop.

En los casos de la paralelización de nodos y en los nodos paralelizables, al implementarlo con Hadoop, se pierde la computación orientada a flujo de datos. Esto se da debido a que al implementar un trabajo MapReduce por nodo, cada nodo debe tener un conjunto fijo de datos (un batch o DataSet de datos). De esta manera cada nodo debe esperar a que sus predecesores terminen su trabajo para comenzar a ejecutar. Esto puede verse mejor en la Figura 14.

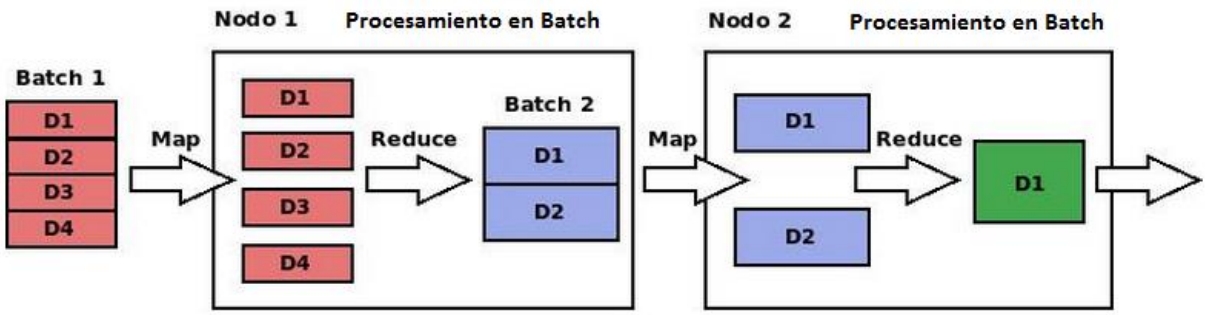

Figura 14 - Modelo básico de cómo paralelizar nodos de un workflow con Hadoop

En la Figura 14 se puede ver como cada nodo debe esperar los batch de datos para comenzar a ejecutar, el Nodo 2 no puede ejecutar incluso cuando una parte del batch, como D1 del batch 2 ya está disponible.

Además de esto, al implementar MapReduce, cada nodo internamente, también tiene un proceso en dos etapas, ya que primero se ejecutan los maps en un batch y luego los reduce que producen el siguiente batch. Con esta forma de procesamiento no se pude lograr la paralelización de flujo.

En el modelo visto en la sección Paralelización de workflows Enteros, la computación orientada a flujos podría conservarse un poco mejor, pero adolecen por lo menos del batch entre las funciones map y reduce. Si bien sería la que más respeta la computación orientada a flujos no es la mejor solución posible al problema como se muestra más adelante.

4.1.1.2 Alta concurrencia de tareas en Hadoop

En cuanto a la alta concurrencia, la cual define que todas las tareas de un workflow científico deberían poder ejecutar simultáneamente entre sí, Hadoop también adolece en la implementación. Por la forma en la que Hadoop puede distribuir el trabajo con el algoritmo MapReduce, simplemente no es posible implementar este requerimiento completamente.

Como se ha visto en la sección anterior, que refiere a la computación orientada a flujos de datos al implementarse con Hadoop, para los dos casos de nodos paralelizables, estos deben ejecutar de forma serial. Si bien cada nodo internamente sería paralelo, nunca los nodos serían paralelos entre sí.

Esto queda claro en la Figura 14, se puede ver cómo para que ejecute el Nodo 2, el Nodo 1 debe dar el batch de datos producto de su ejecución, por lo que este modelo no provee una buena solución al requerimiento de alta concurrencia.

Por otro lado la paralelización entera de workflows, tiene el mismo problema que se describió en la sección anterior, se podría llegar a mostrar cierto paralelismo en algún grado, pero este no sería total.

4.1.2 Workflows Científicos con Storm

En esta sección se estudia cómo Storm se adapta al paradigma de Workflows Científicos. Al igual que se hizo con Hadoop se pondrá un énfasis importante en cuanto a la computación orientada a flujos de datos y a la paralelización de tareas.

4.1.2.1 Flujos de datos en Storm

Teniendo en cuenta la descripción de Storm (ver sección ¿Qué es Storm?), el cumplimiento de este requerimiento por parte de la plataforma queda bastante evidente. La computación orientada a flujos de datos es uno de los casos de uso presentados en la documentación de Storm [59] como procesamiento de flujos.

Utilizando la terminología de Storm, los streams (para más información ver Streams en el anexo de Storm) se mapean exactamente con los flujos de datos. Por lo que Storm cumple con la computación orientada a flujos de datos de hecho.

4.1.2.2 Alta concurrencia de tareas en Storm

Al abordar estos requerimientos con Storm, se abordarán tomando en cuenta tanto la multi instanciación como la paralelización de tareas, requerimientos definidos en Alta concurrencia y Patrones multi instancia.

Antes de empezar a aplicar los conceptos, debemos tener en claro que implican estos requerimientos. Para ello mostraremos lo que se pretende implementar basados en el workflow diagramado a continuación. En la Figura 15 las cajas con nombre “Input X” son los nodos de entrada de los datos, las cajas “NT X” representan los nodos de un workflow y “Output” representa los nodos de salida de un workflow.

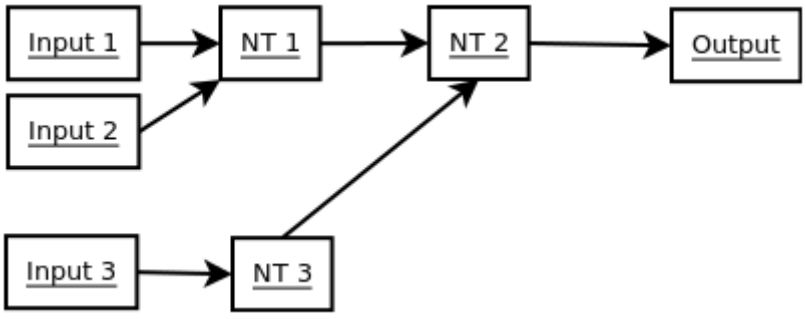

Figura 15 - Workflow básico

Al utilizar Storm se podría implementar este tipo de workflows creando una topología análoga, sustituyendo los nodos “Input X” por spouts. De esta manera por la definición de ejecución de tareas en Storm, estos nodos ejecutan concurrentemente.

4.1.3 Evaluación de herramientas de Big Data

En una primera instancia del proyecto, se evaluó la utilización de Hadoop para distribuir el procesamiento de los workflows, incluso se encontraron varios artículos donde se refiere a cómo hacerlo [56], [57] y [58]. Por lo visto en la sección Workflows Científicos con Hadoop se determinó que Hadoop no es una buena opción y se continuó investigando otras opciones para la distribución del trabajo de los workflows. Luego del análisis realizado en Workflows Científicos con Storm se decidió la utilización de Storm al ser una mejor opción para la representación de los workflows científicos.

4.2 Requerimientos del Proyecto

Luego de hacer un análisis de los distintos SWfMS y el estado del arte de lo que hoy en día es conocido como Big Data se ven oportunidades para combinar estos dos mundos. Una de las principales virtudes de los SWfMS que nos llevan a pensar esto, es su capacidad de paralelización, producto de un modelo de computación orientado a los flujos de datos, así como el manejo y procesamiento de grandes volúmenes de datos.

4.2.1 R1 Taverna como SWfMS

En el contexto del proyecto se evaluaron dos de los más populares SWfMS, Kepler y Taverna. Para no implementar un SWfMS, una decisión prácticamente desde el inicio fue tomar un SWfMS existente, y ejecutar sus workflows con las características de Big Data descritas.

Las características de Taverna que lo hacen idóneo para este proyecto son las siguientes:

- **Un solo modelo de computación:** Taverna solo cuenta con el modelo de computación orientado a flujos de datos, mientras que Kepler cuenta con distintos modelos de computación a través de la implementación de *Directores*. Para simplificar la implementación se considera mejor solo contar con un modelo de computación.

- **Gran número de workflows implementados:** Solo basta entrar a myExperiment (plataforma para compartir workflows científicos [60]) y ver la cantidad de workflows para Taverna y para Kepler. De esta manera se puede encontrar más casos de uso.

- **Usabilidad:** En una primera instancia al equipo de trabajo le pareció más sencillo utilizar Taverna que Kepler, por lo que también agiliza el desarrollo de workflows en ese sentido.

- **Orientado a servicios:** El hecho de que muchos componentes sean orientados a servicios es un atractivo para obtener datos de cualquier índole de forma rápida.

4.2.2 R2 Almacenamiento masivo y distribuido de datos

Teniendo en cuenta lo que ya se ha mencionado de los SWfMS y considerando un escenario de Big Data para el manejo de resultados parciales, entrada y salida. Es necesario contar con un sistema de almacenamiento de gran porte, con altas velocidades, distribuido y agnósticos en tipos de dato. Se requiere la integración de estas tecnologías tanto para el uso interno de la plataforma, como para utilizar dentro del diseño de los workflows.

4.2.3 R3 Paralelización de las tareas de un workflow

Como ya se ha mencionado en la sección de Alta concurrencia los procesos que ejecuten una tarea de un workflow deben ejecutar ni bien tengan datos para procesar. Deben hacerlo de forma concurrente. Este requerimiento es esencial dado que forman parte de la definición de un SWfMS.

4.2.4 R4 Ejecución de workflows de manera distribuida

Más allá de la concurrencia de las tareas de un workflow, definido como requerimiento en el siguiente punto, es de interés del equipo de trabajo dar la posibilidad distribuir dichas tareas en distintas unidades de cómputo (UC). Este requerimiento da la ventaja de poder escalar horizontalmente al agregar más UC. Hoy en día Taverna no brinda esta posibilidad ya que se ejecuta en una sola UC. La forma de cómo se distribuyen los workflows debe ser transparente al usuario, es decir que el usuario no tomara decisiones de cómo se distribuyen los workflows, la plataforma será quien decida cómo hacerlo.

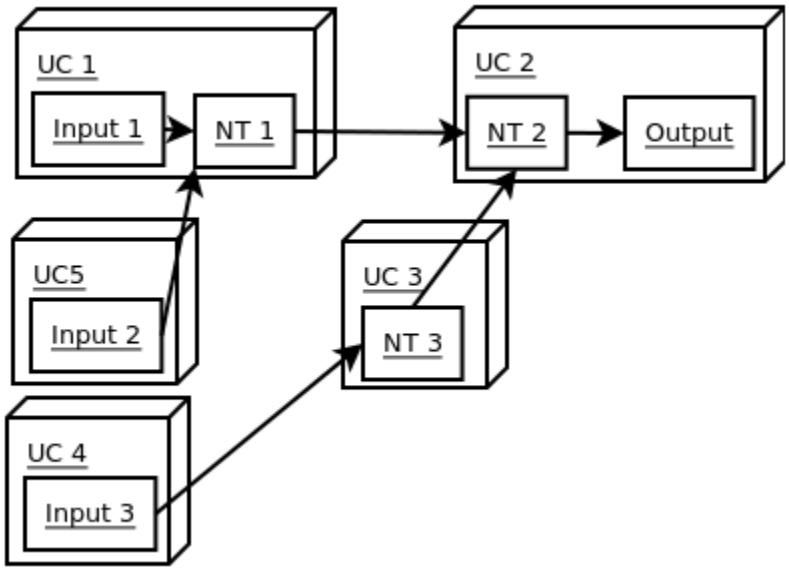

Figura 16 - Workflow distribuido entre distintas UC

En la Figura 16 se puede ver una distribución posible para las tareas del workflow de la Figura 15. En este caso las siete tareas del workflow quedan distribuidas en cinco UC, pudiendo ser otra la combinación posible. De esta manera se aprovechan las diferentes UC del ambiente distribuido, tratando de influir en un incremento de la performance en la ejecución dado que los recursos van a ser mayores en un ambiente de esta naturaleza.

4.2.5 R5 Multi-instanciación de procesadores con ejecución concurrente

Teniendo en cuenta lo visto en la sección de Patrones multi instancia debe ser posible definir la cantidad de procesos que ejecutarán la misma tarea. Estos procesos deben ejecutar concurrentemente. Esto agrega una dificultad extra y es que se debe preservar el orden de los datos en la salida del conjunto de procesos que ejecutan la tarea.

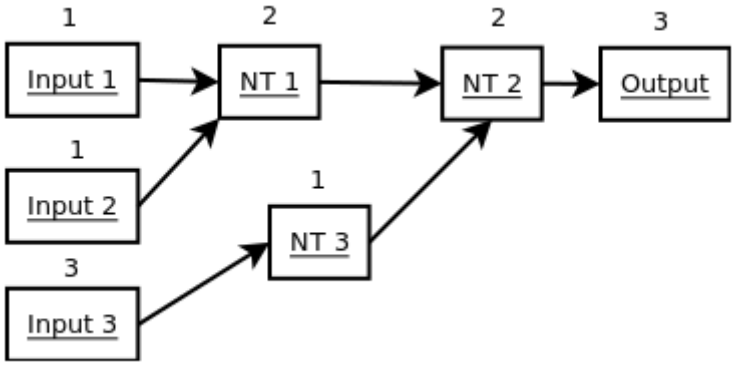

Figura 17 - Workflow con nivel de paralelismo asociado por nodo

En la Figura 17 se muestra cómo cada nodo de un workflow tiene un nivel de paralelización asociado. Cada número representa la cantidad de procesos que ejecutarán la tarea correspondiente al nodo, esto se puede ver en la Figura 18.

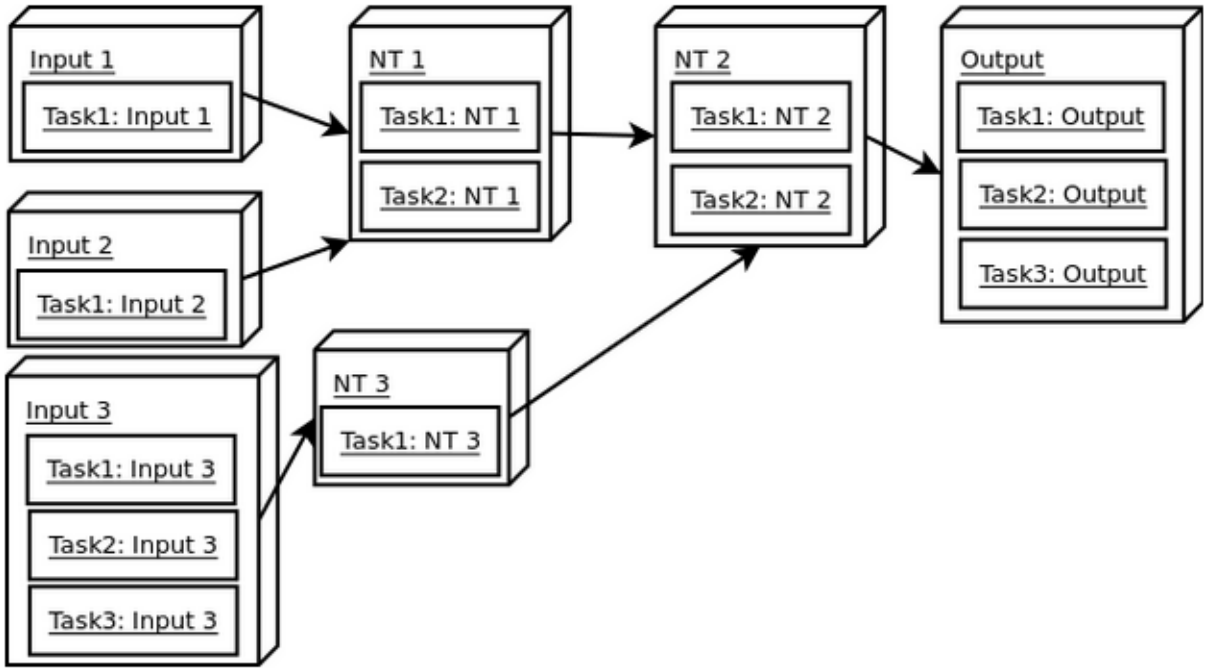

Figura 18 - Workflow distribuido y multi instanciado con por niveles de paralelización asociados

A modo de ejemplo el nodo “NT 2” está configurado para ejecutar en dos instancias, por esa razón “NT 2” se divide en dos tareas, “Task1” y “Task2”. Ambas instancias ejecutan el mismo código, el código correspondiente al nodo “NT 2”.

Una cosa a destacar del diagrama presentado en la Figura 18 es que existe un abuso de notación en las conexiones de las tareas. No se especifica entre que instancias es la conexión del flujo de datos ya que no interesa. Por ejemplo cada dato que produzca la tarea “Task1” del nodo “NT 3” será procesado por “Task1” o “Task2” del nodo “NT 2” sin ninguna consideración especial.

4.2.6 R6 Provenance

Es de interés poder rastrear la ejecución de un dato en el workflow, de esta manera se debe proveer un mecanismo de provenance el cual utilice la indexación para marcar el camino de los datos dentro del workflow.

4.2.7 R7 Consumo de datos de almacenamiento masivo

Teniendo en cuenta que se desea implementar un SWfMS en un escenario de Big Data, es de interés darle al usuario, primitivas para el consumo de datos de Bases de Datos NoSQL.

4.2.8 R8 Pausa y re ejecución de workflows

Considerando que el tiempo ejecución de los workflows por lo general es grande y que durante una ejecución los resultados se van guardando (ver R6 Provenance), se considera de valor la posibilidad de “pausar” y reanudar la ejecución de un workflow. Esto permitiría al usuario consultar y modificar los valores parciales de la ejecución sin tener que esperar a la finalización.

4.2.9 R9 Storm

Teniendo en cuenta lo visto en la sección Hadoop vs Storm se llegó a la conclusión de que Storm es la mejor opción a la hora de implementar un SWfMS. Storm permite construir topologías de procesamiento de flujos de datos distribuidas y concurrentes, esto permite cumplir con los requerimientos R3 Paralelización de las tareas de un workflow y R4 Ejecución de workflows de manera distribuida. Además permite definir la cantidad de instancias de cada nodo de la topología por lo que permite cumplir con R5 Multi-instanciación de procesadores con ejecución concurrente de forma sencilla. Por todo esto es que se decide utilizar cómo tecnología de procesamiento de datos a Apache Storm.

4.3 Multi-instanciación, distribución y paralelismo

Estos tres requerimientos tienen algo en común y es que apuntan básicamente a mejorar la capacidad de procesamiento. Esto es a través de la concurrencia, ya sea de diferentes nodos o de las múltiples instancias de un nodo, y de la distribución de los procesos.

Es la implementación de estos tres requerimientos en conjunto uno de los principales objetivos del proyecto para lograr una buena capacidad de procesamiento. Este objetivo es lograr paralelizar los workflows no solo a nivel de nodos, sino a nivel de las múltiples instancias (“Task”) de los nodos. A continuación en la Figura 19 se presenta un ejemplo basado en el workflow definido previamente en el que supondremos que cada “Task” asociada a los nodos Taverna emite sólo una vez, un solo valor, y lo mismo hacen el resto de las “Tasks” del workflow. El camino de cada una de estas tuplas es representado por un color diferente.

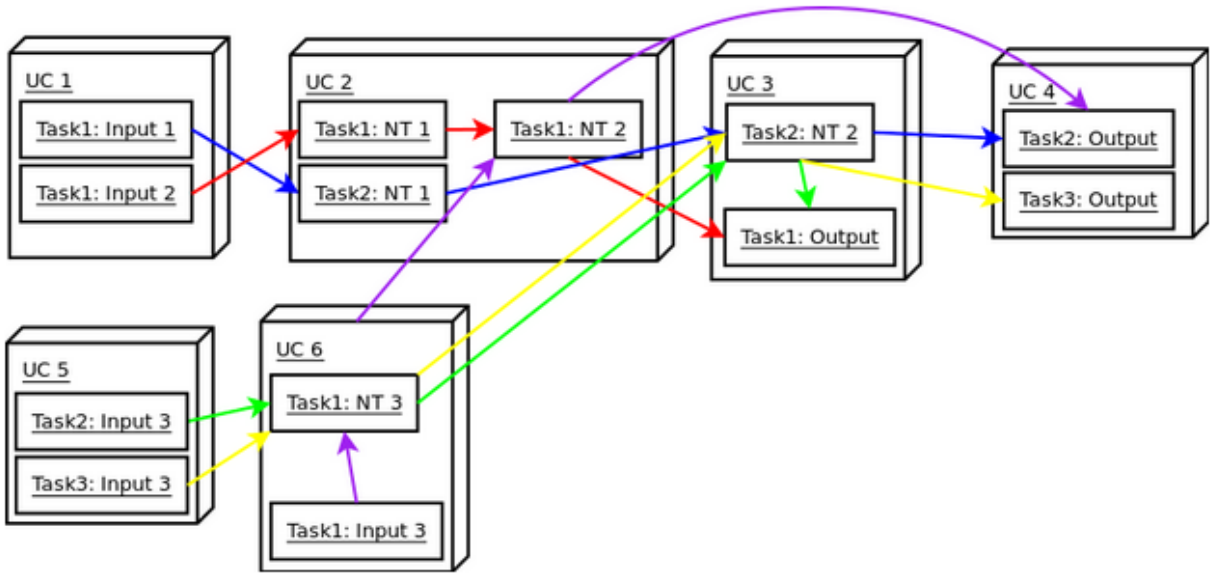

Figura 19 - Multi instanciación y paralelismo con algunos caminos de posibles tuplas en el workflow

4.4 Comparación de modelos: Taverna vs Storm

Si bien la elección de Storm soluciona muchos problemas, y la selección de Taverna soluciona otros muchos, la conjunción de ambas herramienta genera nuevos requerimientos. Estos requerimientos surgen de:

- Las posibilidades de manejos de entrada que tiene Taverna en cada nodo.
- Tener que generar mecanismos para mantener el orden de los flujos de datos (problema que surge por la multi instanciación y distribución).
- El hecho de que una topología Storm se mantiene siempre en ejecución, a diferencia de un workflow que tiene un inicio y un fin.

En esta sección se explican los requerimientos que surgen de implementar un SWfMS con Storm usando cómo SWfMS base a Taverna.

4.4.1 Manejo de entradas de nodos

En Taverna existe la forma de combinar los valores de entrada en cada nodo. Existen dos formas de combinar las entradas, usando un producto vectorial o usando un producto cartesiano. El producto vectorial combina los valores de las distintas entradas uno a uno en el orden que llegan. El producto cartesiano en cambio combina los valores de las distintas entradas de uno nodo “todos contra todos”. En la Figura 20 se puede ver un ejemplo de ambos.

En Storm tener links entre un nodo y varios predecesores, representa que una tupla puede venir por un link o por el otro. En Taverna es un poco diferente, ya que si un nodo tiene dos

links entrantes, esto representa que la tupla que procesa ese nodo será la combinación de todos los valores, determinado por los distintos productos. En la Figura 20 se muestran estas diferencias.

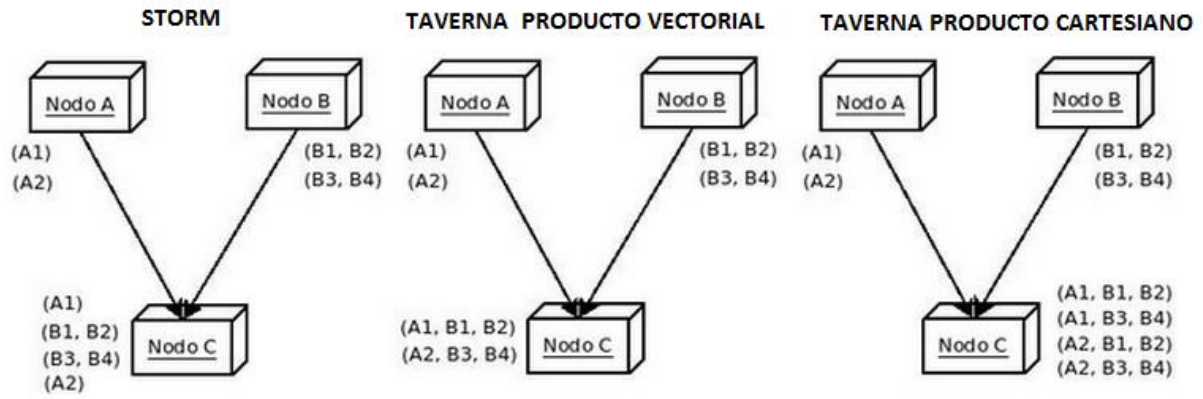

Figura 20 - Combinación de datos en Storm, Taverna con producto vectorial y cartesiano

En Figura 20 se puede ver un nodo C, con dos nodos A y B predecesores, en la primer imagen se presenta el modelo de Storm, dado que los nodos emiten, dos tuplas cada uno, A emite (A1), (A2), y B emite (B1, B2), (B3, B4), el nodo C termina recibiendo cuatro tuplas, en cualquier orden (en este caso se ilustró solo una combinación posible).

En la segunda imagen se muestra el modelo Taverna con producto vectorial, en ese caso el nodo C recibe combinaciones uno a uno en el orden emitido por los nodos predecesores. En la tercera, se presenta también el modelo Taverna, pero con producto cartesiano, el cual combina todos los valores de un link, con los valores del resto.

Esto implica que se deben crear mecanismos para ejecutar el comportamiento del producto cartesiano y el producto vectorial al comienzo de cada nodo de la topología.

4.4.2 Indexación y ordenamiento de tuplas

Teniendo en cuenta de que existirán varias instancias de cada nodo, no hay nada que garantice que se mantenga el orden de las tuplas a medida que avanzan en la topología. Es necesario mantener el orden de las tuplas en algunas circunstancias, entre ellas al momento de generar la salida o al momento de manejar las tuplas con un producto vectorial.

Es por esto que se deben crear mecanismos de generación de índices que indiquen la posición de una tupla dentro del flujo. En base a estos índices se ejecuten procesos de ordenamiento.

4.4.3 Detección de finalización de workflow

Una de las diferencias entre el modelo de ejecución de Storm y los workflows científicos es que en una topología está siempre activa hasta que el usuario decida desactivarla. En cambio un workflow científico está activo mientras haya datos de entrada para procesar. Esto implica que se debe implementar un mecanismo que detecte cuando se han terminado de procesar los datos y finalice automáticamente la topología.

4.4.4 Traducción de modelos

Teniendo en cuenta los requerimientos anteriores, es necesario crear un sistema que sea capaz de convertir un workflow de Taverna en una topología Storm. En esta sección se toma un workflow simple como ejemplo y se muestra la topología que debe ser construida. El modelo del proyecto se muestra con las entidades de Storm. Se quiere construir un workflow a partir de los datos obtenidos de un servicio REST y otro parámetro de entrada para luego ser procesados por un beanshell como se ve en la Figura 21 “Diseño conceptual del workflow”.

Este diseño conceptual se pasa al diseñador de workflows de Taverna en donde casi no hay diferencias como se ve en la Figura 21 “Diseño de workflow en Taverna”.

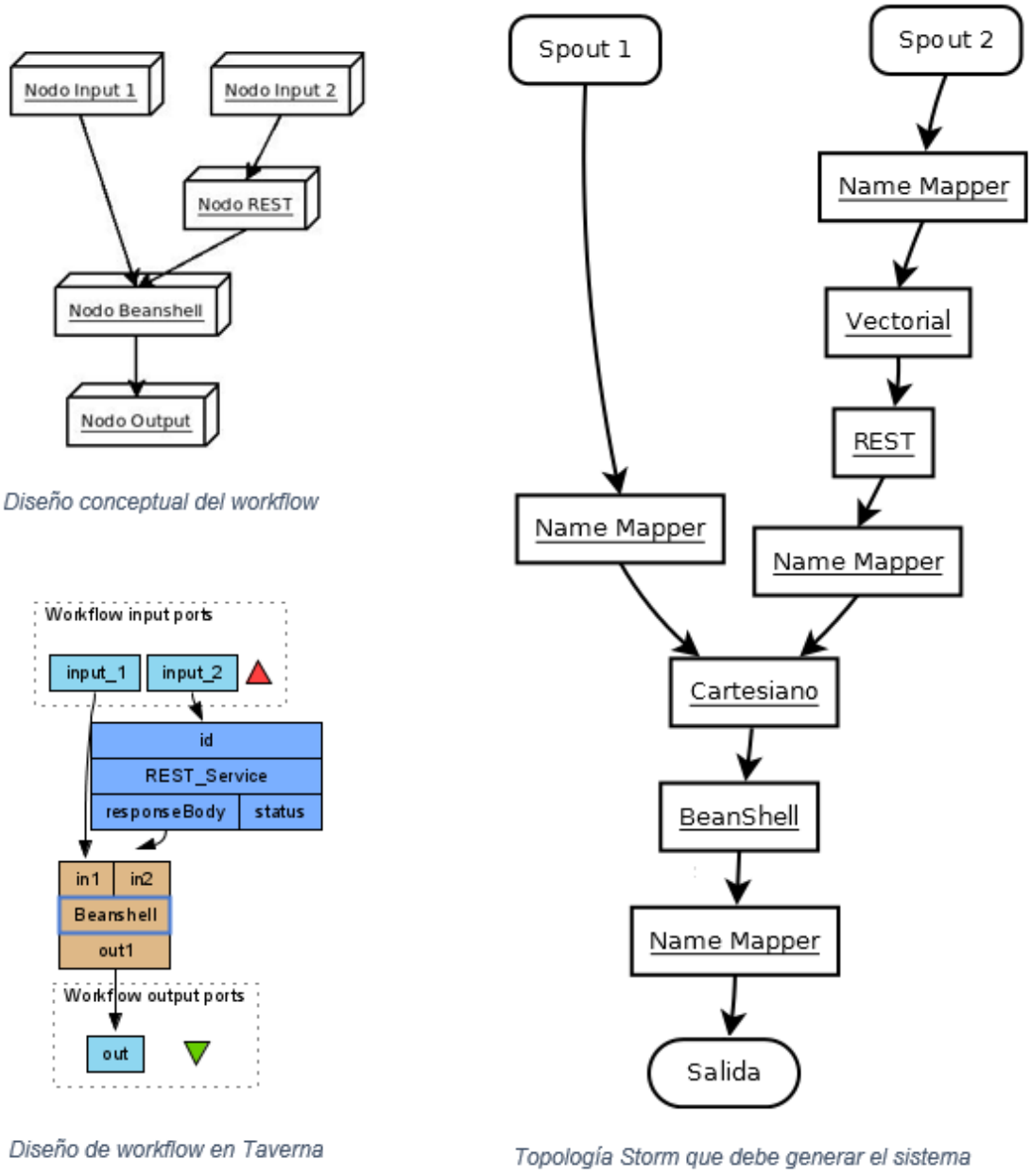

Figura 21- Traducción de modelos

En la Figura 21 “Topología Storm que debe generar el sistema” se muestra el modelo de ejemplo de la topología que debe generar el sistema.

A continuación se explican las equivalencias entre los modelos. Tomando como base el modelo en Storm y comparándolo con Taverna.

- **Spout 1:** es equivalente al input_1.
- **Spout 2:** es equivalente al input_2.
- **Los Name Mapper:** son equivalentes a los links entre entidades en Taverna

- **Vectorial:** Aunque el modelo de Taverna no los muestra la entrada input_2 es mapeada a id usando un producto vectorial por eso es necesario en el modelo del framework un producto vectorial antes del REST.
- **REST:** es mapeado con el REST.
- **Cartesiano:** al igual que el vectorial en el modelo Taverna no se ve el producto escalar pero la relación entre in1 e in2 es resuelta mediante un producto escalar
- **Beanshell:** es mapeado con el Beanshell
- **Salida:** es equivalente a out, además tener en cuenta que este nodo es un ordenador ya que debe escribir la salida de forma ordenada.

Para entender mejor como se mapean los workflows ver el anexo Aplicando Storm sobre los requerimientos.

4.5 Alcance

El alcance del proyecto abarca los siguientes requerimientos:

- R1 Taverna como SWfMS.
- R2 Almacenamiento masivo y distribuido de datos.
- R3 Paralelización de las tareas de un workflow.
- R4 Ejecución de workflows de manera distribuida.
- R5 Multi-instanciación de procesadores con ejecución concurrente.
- R6 Provenance.
- R9 Storm.

Además todos los requerimientos descritos en la sección Comparación de modelos: Taverna vs Storm.

Se decidió dedicarse a estos requerimientos y los restantes dejarlos para una siguiente iteración del proyecto.

5 Diseño

El siguiente capítulo se enfoca en describir la solución propuesta tomando como punto de partida los requerimientos establecidos en la sección de Requerimientos del Proyecto.

5.1 Visión general de la solución

La arquitectura está constituida por dos componentes, uno que se encarga de la ejecución del flujo de datos, y otro que realiza la comunicación especificando los workflows científicos e interactuando con el componente que los ejecuta. Se diseñó una arquitectura para poder implementar una herramienta de workflows científicos tomando como base la API REST de Taverna Server para realizar la comunicación y a Storm para la ejecución de los workflows.

El diseño de la solución propuesta está pensado para poder realizar las mismas operaciones provistas por la API REST de Taverna.

La solución realiza el despliegue de las topologías definidas en Taverna Workbench, así como la comunicación con las topologías luego de desplegadas para proveer información de estado de las mismas y otras funcionalidades.

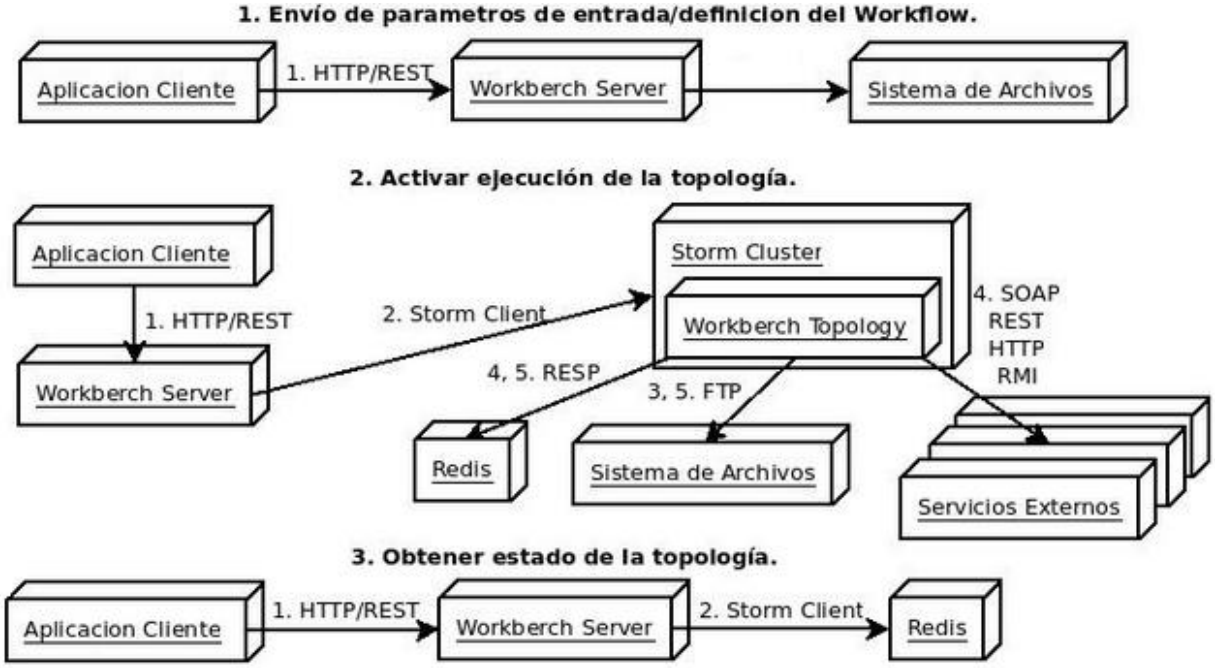

Figura 22 - Vista de solución y procesos de integración

En la Figura 22 se puede ver una vista general de la solución en tres de los procesos principales de la aplicación. Estos procesos son el envió de parámetros y definición de

workflows (1), la activación de la ejecución (2), y la obtención del estado de ejecución de la topología (3).

Envió de parámetros de entrada/definición del workflow (1)

1. El usuario o aplicación cliente realiza una operación a través de la API REST para enviar los parámetros de entrada o la definición del workflow al Workberch Server.

2. El Workberch Server guarda los archivos en el sistema de archivos para que las topologías puedan tomar los datos y la definición de algún lugar. En el diagrama se muestra FTP como protocolo de comunicación con el sistema de archivos, pero se deja al administrador la decisión del protocolo de comunicación con el sistema de archivos.

Activar ejecución de la topología (2)

1. Una vez que los archivos necesarios para la topología han sido enviados con el proceso (1), la aplicación cliente puede enviar una señal que activa la ejecución de la topología a través de la API REST.

2. Cuando el Workberch Server recibe a través de la API REST el comando del punto 1, envía a través del Storm Client el componente Workberch Topology para que ejecute en el clúster Storm con los parámetros y definiciones enviados en el proceso (1).

3. En este punto el Workberch Topology comienza a crear la definiciones de la topología a través de la definición provista en (1), una vez que la topología está armada se la envía a ejecutar.

4. En este punto la topología ya comenzó a ejecutar y los distintos nodos comenzarán a interactuar con los servicios externos que cada nodo requiere. Además de esto también se va guardando toda la información de control y de provenance necesaria en Redis.

5. Este punto se identifica por el momento en el que el workflow termina de ejecutar. Una señal de finalización se transmite a través de todo el workflow cuando la ejecución va terminando por cada nodo. Cuando los nodos asociados a los puertos de salida reciben esta señal escriben los valores finales en el sistema de archivos para que el usuario pueda descargar la salida. Además se escribe en Redis un parámetro de control que determina la finalización de todo el workflow el cual es necesario para (3).

Obtener estado de la topología (3)

1. Luego de que el usuario en (2) ejecuta una topología, la aplicación cliente puede consultar en cualquier momento el estado de ejecución a través de la API REST.

2. Cuando el Workberch Server recibe un pedido de estado del workflow, lee determinadas variables de control en Redis para saber si el workflow está ejecutando. Como se vio en el paso 5 de (2) esta variable se activa al final de la ejecución, por lo que el resultado de esta operación es devuelto dependiendo el valor de esta variable de control.

En la Figura 22 quedan representados todos los elementos de la solución así como todas las comunicaciones y protocolos utilizados entre ellos.

5.2 Arquitectura

En esta sección se describe la arquitectura del sistema utilizando un conjunto de vistas complementarias, donde cada una define un aspecto del diseño del sistema.

5.2.1 Vista de componentes

La arquitectura consta de dos componentes principales: Workberch Server y el Workberch Topology. Estos utilizan las herramientas Storm y Redis para poder realizar sus tareas. En esta sección se muestra cómo los componentes interactúan entre sí y de la forma que son utilizados los clusters Redis y Storm.

5.2.1.1 Workberch Server

El componente Workberch Server tiene la responsabilidad de implementar la API REST de Taverna además de comunicarse con Storm y Redis para ejecutar las topologías.

En la Figura 23, se muestra cómo el controlador REST, el cual implementa el API de Taverna, interactúa con Storm y Redis para realizar las distintas operaciones del servidor, en ambos casos se utiliza estos componentes a través de un cliente.

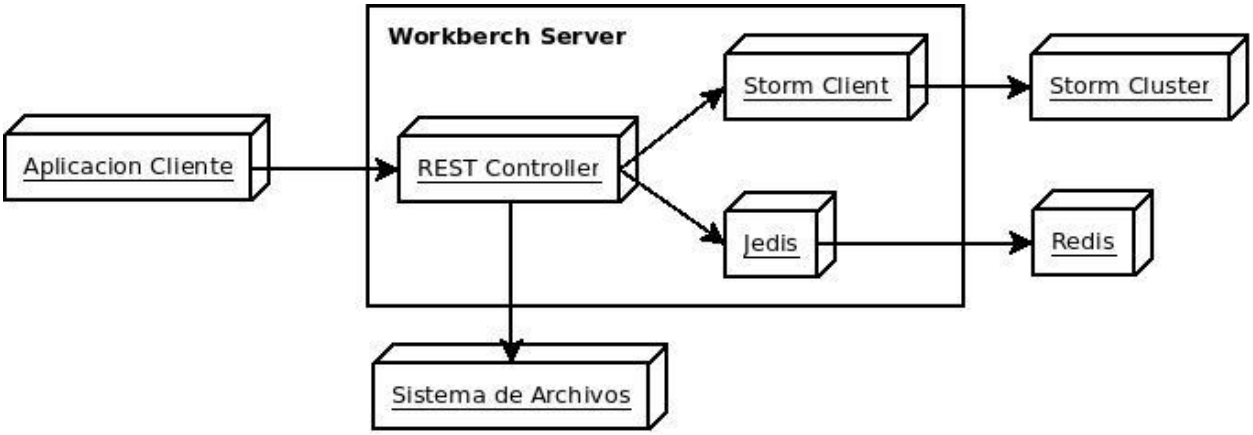

Figura 23- Diagrama de componentes de Workberch Server

Cuando el servidor necesita enviar la topología a ejecutar en Storm, ejecuta una operación a través del cliente Storm enviando la topología y determinados parámetros de ejecución.

5.2.1.2 Workberch Topology

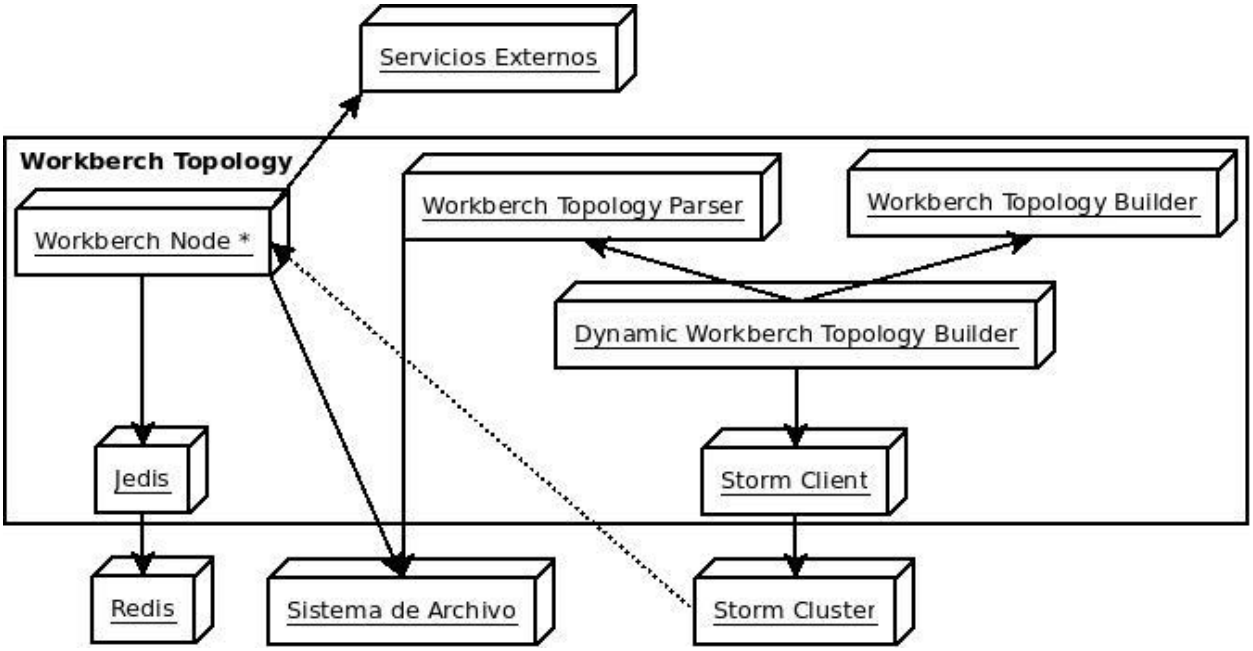

Figura 24 - Diagrama de componentes de Workberch Topology

Este componente que se muestra en la Figura 24 tiene la responsabilidad de construir la topología Storm en base al workflow definido en Taverna. Esto es responsabilidad de “Dynamic Workberch Topology Builder” que básicamente utiliza dos componentes: “Workberch Taverna Parser” y “Workberch Topology Builder”. El primero se encarga de leer el archivo XML del workflow definido en Taverna construyendo una representación interna del workflow. Luego el “Workberch Topology Builder” utiliza esta representación interna para transformarla en una topología Storm.

Todos los nodos Storm (“Workberch Node”) se comunican con la base de datos: Redis tanto para el manejo de índices cómo para el mantenimiento del Provenance. Cabe destacar que los Workberch Node fueron pensados para que sea sencillo agregar nuevos nodos que realicen otras funcionalidades (aparte de Beanshell, Text Constatant, XPath, etc.) ya que por razones de tiempo no se implementaron todos los nodos primitivos de Taverna.

En la sección Workberch Topology Builder del capítulo Implementación, se puede ver en más profundidad los subcomponentes de este componente.

5.3 Despliegue

A continuación se hará una descripción de cómo se distribuyen y comunican los distintos elementos de la solución. El clúster Storm está constituido por un Nimbus, un Zookeeper y los Supervisors. Más información sobre cómo está compuesto un cluster Storm en Anexo Componentes de un cluster Storm.

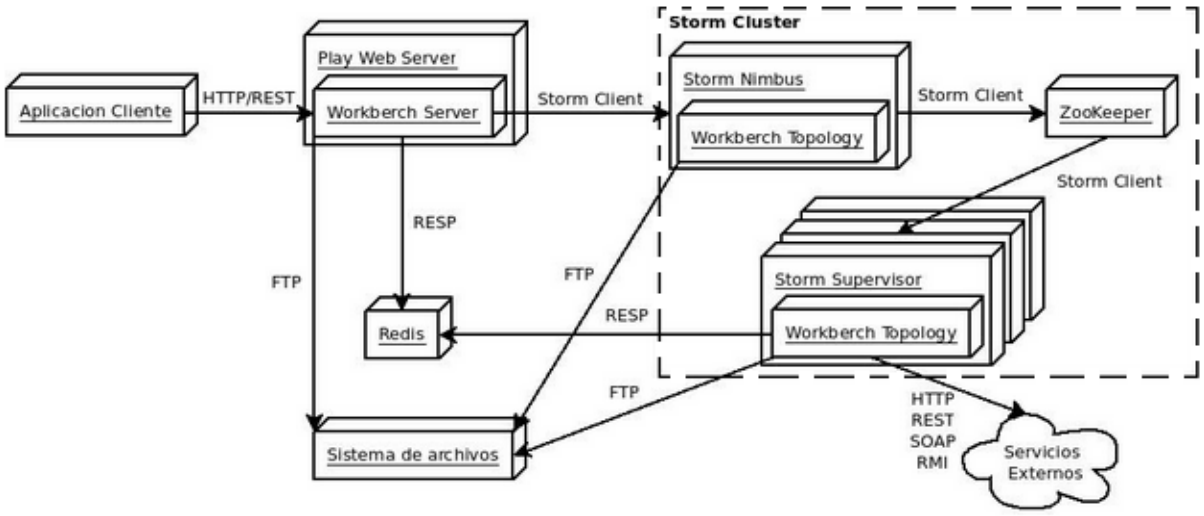

Figura 25 - Diagrama de despliegue

Partiendo de la Figura 25 se realiza una descripción de los componentes:

1. Workberch Server envía la topología a Storm a través de Nimbus.

2. Nimbus se comunica con Zookeeper para enviar la topología a los Supervisors.

3. Zookeeper se comunica con los Supervisor para enviar la topología.

4. Workberch Topology utiliza Redis para salvar y consultar información de los workflows.

5. El Sistema de archivos distribuido es accedido por Workberch Topology para acceder a los archivos de las topologías y también los archivos de entrada/salida.

6. El Sistema de archivos distribuido es accedido por Workberch Server para acceder a los archivos de las topologías y también los archivos de entrada/salida.

7. Workberch Server utiliza Redis para salvar y consultar información de los workflows.

Las comunicaciones (2) y (3) son descritas en profundidad en el anexo Componentes de un cluster Storm.

5.4 Procesos

En esta sección se describirán algunos de los procesos más relevantes de la arquitectura. Al estudiar la herramienta Taverna se encontraron algunas características importantes en su comportamiento al ejecutar los workflow que se decidió agregar a la solución propuesta. En particular la forma de manejar los productos vectoriales y cartesianos. Es recomendable para el lector repasar los anexos Componentes de un cluster Storm y Definiendo procesamiento en Storm ya que se utilizan conceptos de la tecnología para explicar esta sección.

5.4.1 Indexación y distribución de tuplas

Para simular el comportamiento de Taverna un nodo de procesamiento debe recibir las tuplas de forma tal que dado dos links de entrada, reciba una tupla con la combinación definida (sea vectorial o cartesiana).

A nivel de diseño, interesa que para las unidades lógicas definidas por el workflow, los mecanismos de cómo se reciben y se emiten las tuplas deben mantenerse transparentes. Por esta razón es de interés desacoplar en todo lo que se pueda los mecanismos del manejo de las tuplas.

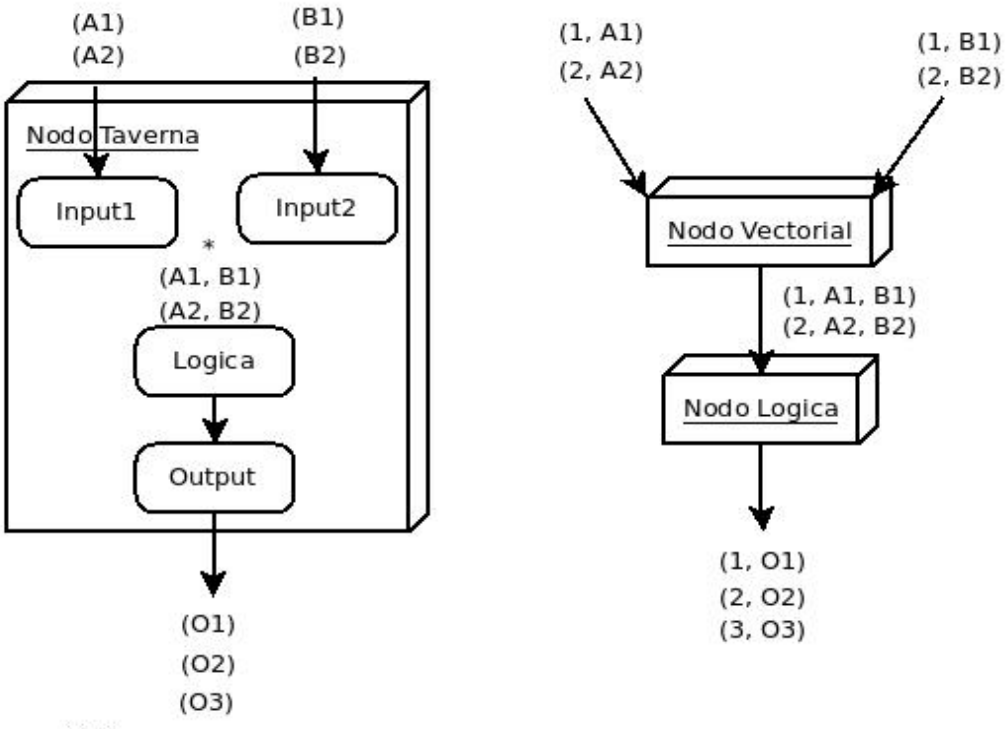

Figura 26 - Producto Vectorial

Debido a esto, se decidió tomar el procesamiento para lograr el producto vectorial y el producto cartesiano y colocarlo como un nodo predecesor obligatorio al nodo que ejecuta la lógica definida. Esto implica que un nodo de Taverna no se corresponde unívocamente con un bolt de Storm si no que se corresponde con una sub red de bolts de Storm.

Esto queda representado por la Figura 26 y Figura 27, a la izquierda se presenta el nodo Taverna (vectorial en el Figura 26 y cartesiano en Figura 27) y del lado derecho la nueva configuración de nodos que representa el nodo Taverna en Storm. Como se pueden ver, en ambos casos los nodos fueron divididos en dos bolts Storm, en el primer nodo se combinan las tuplas y en el segundo se toma la tupla combinada para ejecutar la lógica definida en el nodo.

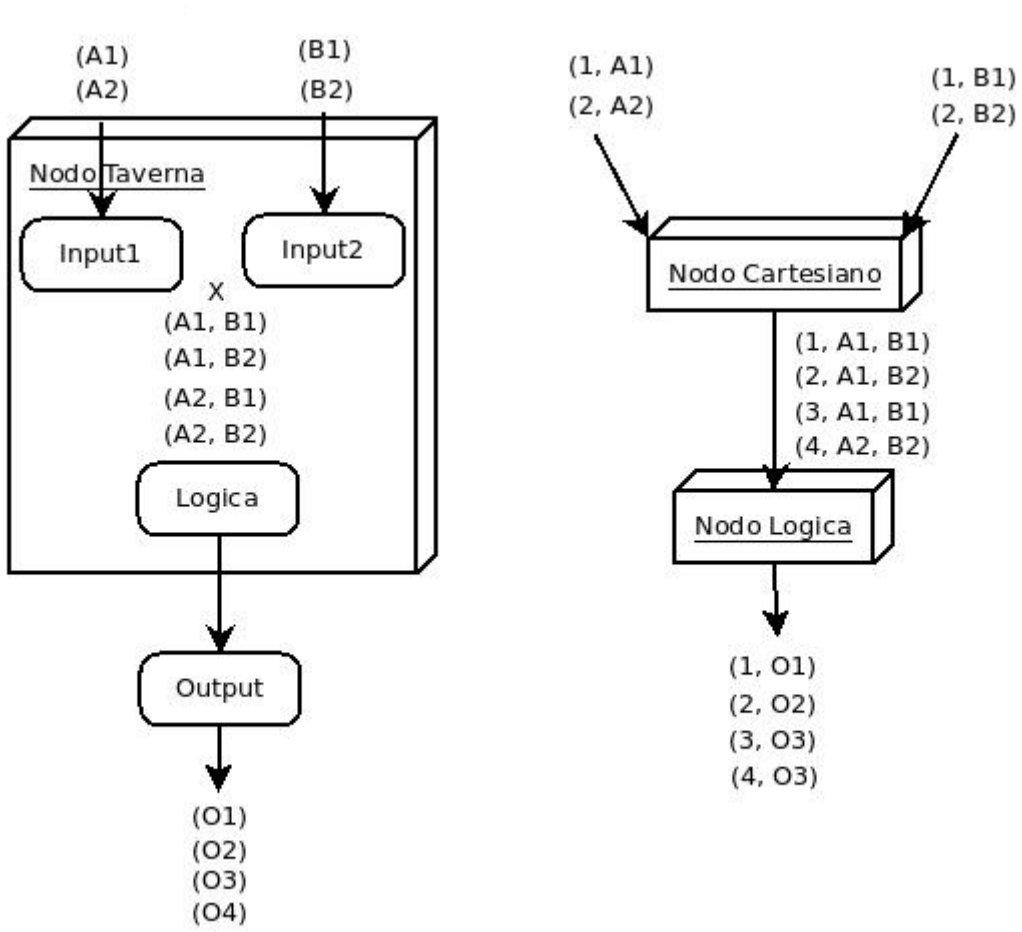

Figura 27 - Producto Cartesiano

En el caso del producto vectorial el nodo Taverna espera cada tupla como una combinación uno a uno de todos los links de entrada. Para simular este comportamiento el nodo predecesor con la lógica del producto vectorial debe recibir tuplas de múltiples nodos y emitir una tupla cuando se tengan dos valores relacionados.

Para lograr el comportamiento anterior se debe mantener las tuplas indexadas de forma de poder saber que tupla desde una fuente se corresponde con que otra tupla de otra fuente.

Se puede apreciar en la Figura 20 de la sección Manejo de entradas de nodos como sería el caso si no se indexara.

Para más información sobre la indexación ver el anexo Diseño detallado de procesos de la solución.

5.4.2 Ordenamiento

Vista la sección *Indexación y distribución de tuplas*, se determina que se deben implementar puntos de ordenamiento en el workflow. Pero no solo necesitamos puntos de ordenamiento para nodos particulares, se necesitan puntos de ordenamiento para determinadas situaciones que se pueden dar por la indexación, para más información ver el anexo Diseño detallado de procesos de la solución.

5.4.2.1 Ordenamiento lineal

El ordenamiento lineal es cuando el ordenador recibe un flujo de tuplas indexado linealmente (como el resultado de un nodo vectorial o luego de un spout) y lo emite en forma ordenada como muestra la Figura 28.

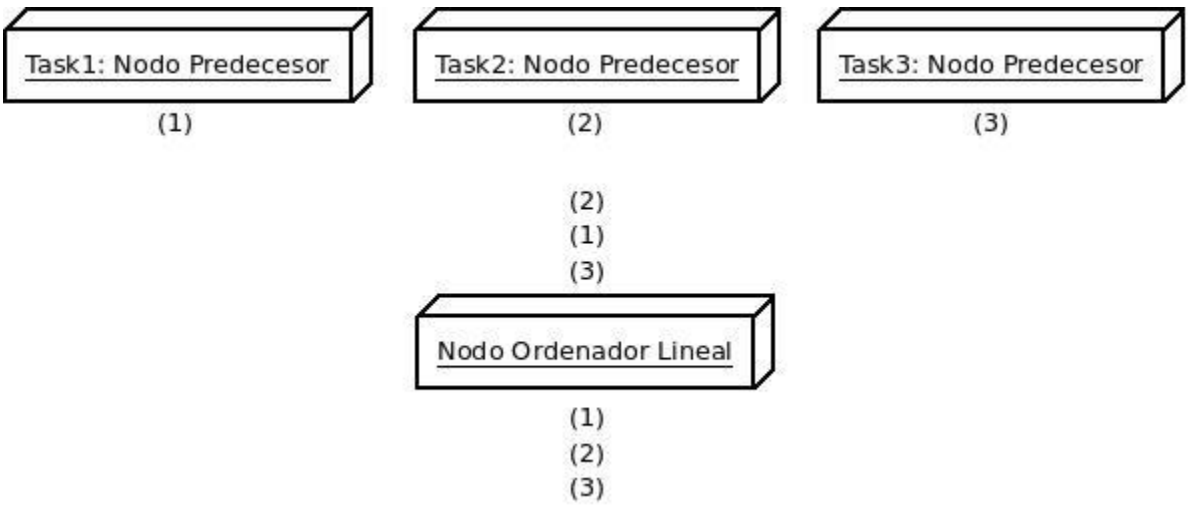

Figura 28 - Ordenamiento lineal

En la Figura 28 se muestra un nodo multi instanciado en tres instancias, para ejemplificar se obviaron los valores y solo se muestran los índices que emite cada instancia. El nodo ordenador recibe los índices desordenadamente (2,1,3) y los emite y procesa de la forma (1,2,3). Para implementar este ordenamiento se diseñó el siguiente algoritmo.

```
hashmap indexMap {
        key: indice,
        value: tuple
}
lastIndex = 1

execTuple(tuple) {
        currentIndex = tuple.getIndex()
        if (currentIndex > lastIndex)
                indexMap.put(currentIndex, tuple)
        else if (currentIndex = lastIndex)
                indexMap.put(currentIndex, tuple)
                do
                        sendTuple = indexMap.get(lastIndex)
                        executeOrdered(tuple)
                        lastIndex++
                while indexMap.containsKey(lastIndex)
}
```

Algoritmo 1 - Ordenamiento lineal

5.4.2.2 Ordenamiento anidado

El ordenamiento anidado es cuando el ordenador recibe un flujo de tuplas anidadas (como el resultado de un nodo cartesiano) y lo emite de forma ordenada, con un índice lineal, como muestra la Figura 29.

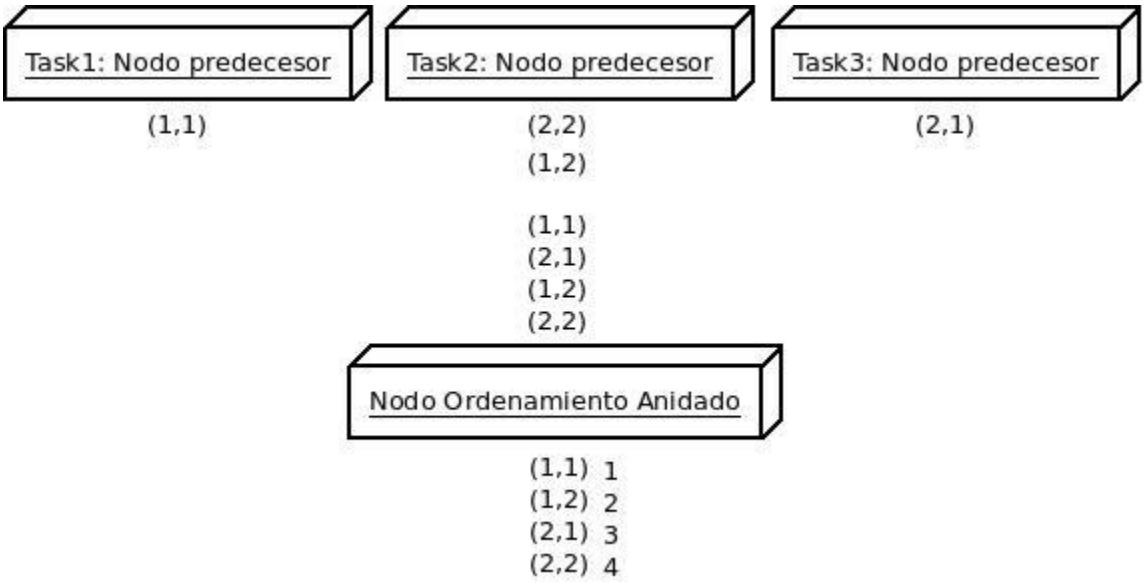

Figura 29 - Ordenamiento anidado de un nivel

En la Figura 29 se puede ver un nodo predecesor que emite los valores anidados (1,1) (1,2) (2,1) y (2,2), al igual que en la Figura 28 solo se muestran los valores de los índices para simplificar. En la salida del nodo de ordenamiento se muestra como se emiten ordenadamente los índices anidados, y a la derecha se muestra el nuevo índice plano, ya que el ordenador anidado cambia el índice anidado por un índice lineal.

El algoritmo para implementar el ordenamiento consta de dos etapas. En la primera etapa (representado en algoritmo por la función *execTuple*) se toma la tupla cuando llega y se la coloca en un mapa (en el algoritmo en la estructura *indexMapList*) que tiene como clave los índices anidados en forma de árbol. En el caso de recibir la última tupla (al chequear la variable *lastValue* en *true*) se entra en la segunda fase del algoritmo.

En esta fase (representada por la función *getPlainMapIndex*) se toma la estructura generada en la fase anterior y una tupla dada por la función *template* la cual devuelve una tupla con la estructura de las tuplas recibidas. Esta tupla (en el algoritmo representado por *templateTreeIndex*) se utiliza para hacer una búsqueda exhaustiva en la estructura generada en la fase anterior, generando los índices en forma de árbol en el orden que deberían ser emitidos. Por cada uno de estos índices se chequea si existe en la estructura previamente generada. En caso de no existir se va un nivel anterior en el árbol generado y se vuelve a chequear, cuando no encuentra un árbol para el valor dado, el algoritmo termina habiendo emitido todos los valores en el orden deseado.

A continuación se puede ver el pseudocódigo del algoritmo descrito anteriormente.

```
hashmap indexMapList{
        key: tree,
        value: tuple
}
hashmap indexMap {
        key: indice,
        value: tuple
}
lastIndex = 1

execTuple(tuple) {
        indexMapList.put(tuple.getIndexTree(), tuple)
        if (lastTuple)
                indexMap = getPlainMapIndex(indexMapList, tuple.getIndexTree().template())
                do
                        sendTuple = indexMap.get(lastIndex)
                        executeOrdered(tuple)
                        lastIndex++
                while indexMap.containsKey(lastIndex)
}

getPlainMapIndex(indexMapList, templateTreeIndex) {
        getPlainMapIndexRec(indexMapList, templateTreeIndex.getNodes(), emptyTree)
```

```
}

getPlainMapIndexRec(indexMapList, listNodes, currentTree) {
	if (listNodes.isEmpty())
		if (indexMapList.containKey(currentTree))
			indexMap.put(lastIndex, indexMapList.get(currentTree))
			lastIndex++
			return true
		return false
	else
		indexValue = 0
		existValue = false
		existSomeValue = false
		do
			currentTree.addNode(indexValue)
			existValue = getPlainMapIndexRec(listNodes.tail(), indexMapList,
							currentTree)
			currentTree.removeNode(indexValue)
			indexValue++
			existSomeValue = existValue or existSomeValue
		while existValue
		return existSomeValue
}
```

Algoritmo 2 - Ordenamiento anidado

También es importante notar, que los nodos ordenadores anidados, deben acumular todos los valores del flujo antes de poder comenzar a emitir, ya que deben hacer una búsqueda exhaustiva en el flujo, lo cual produce un punto de parada del tipo MapReduce visto en Hadoop.

Además tener en cuenta que para esta versión si bien se refiere a *tree* como árbol de índices, son listas y por eso algunas de las simplificaciones del algoritmo.

Este algoritmo queda implementado para índices anidados de un solo nivel, ¿pero qué sucede con el anidamiento en múltiples niveles? En la Figura 30, se muestra como sería la resolución de índices anidados de más de un nivel análogo a lo mostrado en la Figura 29 para un nivel.

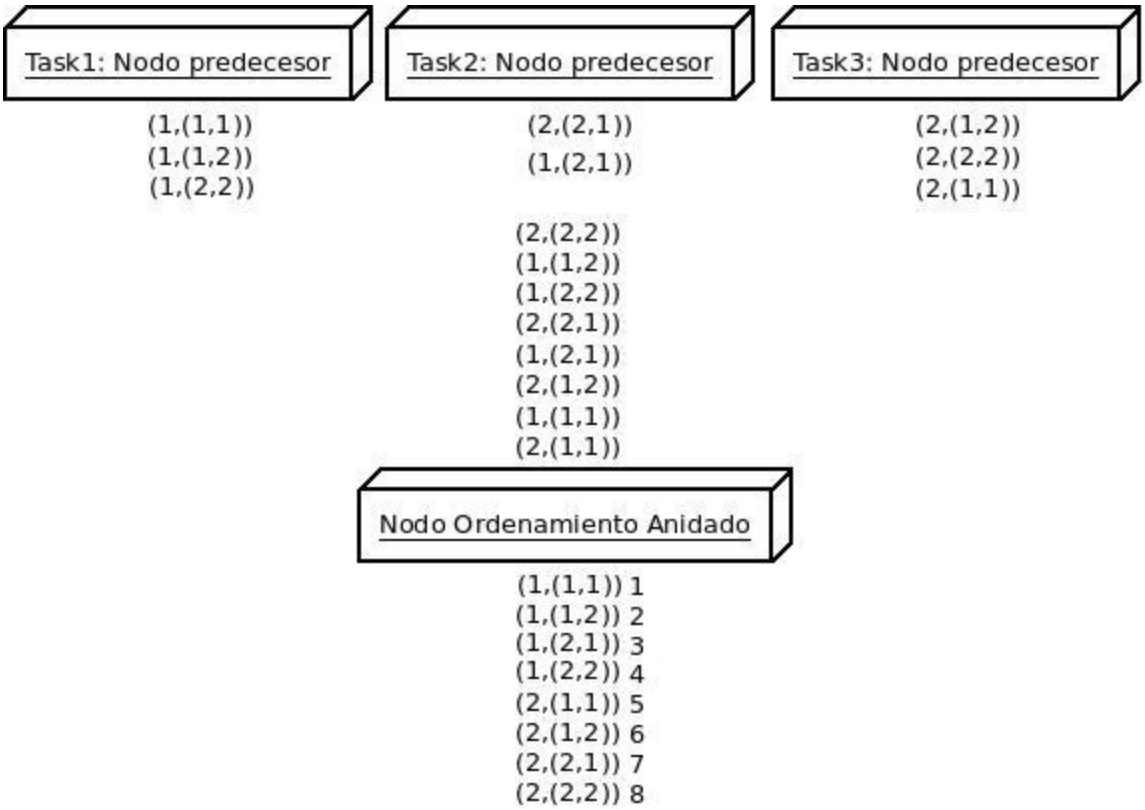

Figura 30- Ordenamiento anidado de más de un nivel

Para simplificar la solución a desarrollar, se decidió que el algoritmo de índices anidados de más de un nivel no sería implementado, ya que no apremian los tiempos del proyecto y no se cree que agregue un valor agregado diferencial.

Para simplificar la problemática, se decidió que el ordenador vaya luego de cada producto cartesiano, de esta manera solo se obtendrán índices anidados de un solo nivel. Esto no será un gran problema, ya que en su mayoría, el producto cartesiano es usado para combinar nodos que emiten una sola tupla con flujos de datos.

En la Figura 31 se muestra como un producto cartesiano es mapeado a nodos Storm, y donde se colocaría el ordenador para evitar índices de múltiples niveles. Al colocar el ordenador en esa posición respecto al producto cartesiano, cada índice anidado creado, será inmediatamente transformado a un índice línea.

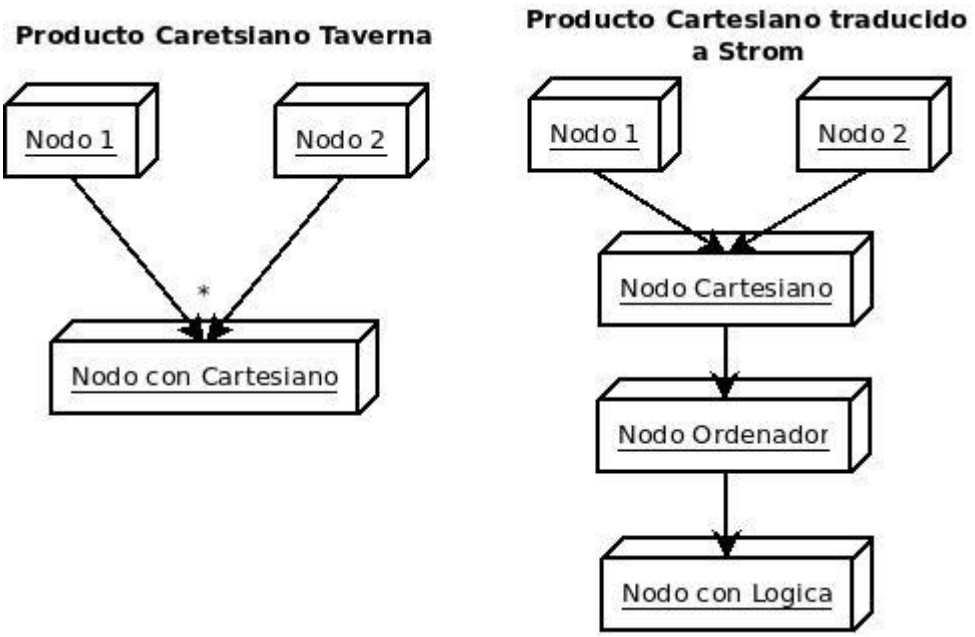

Figura 31 - Relación entre producto cartesiano definido en Taverna y estructura Storm

Además del poco uso del producto cartesiano, los casos en donde se usa son para combinar parámetros unitarios de entrada, pueden mejorarse en sí, para así reducir aún más los cuellos de botella en el ordenamiento.

Por esta razón se diseñará un bolt particular que tomara esos nodos que emiten un solo valor, e implementa una combinación de los valores sin generar índices anidados, siendo más parecido a la mecánica del producto vectorial. A continuación se muestra en la Figura 32 un ejemplo de lo descrito en Taverna y cómo se distribuiría en Storm.

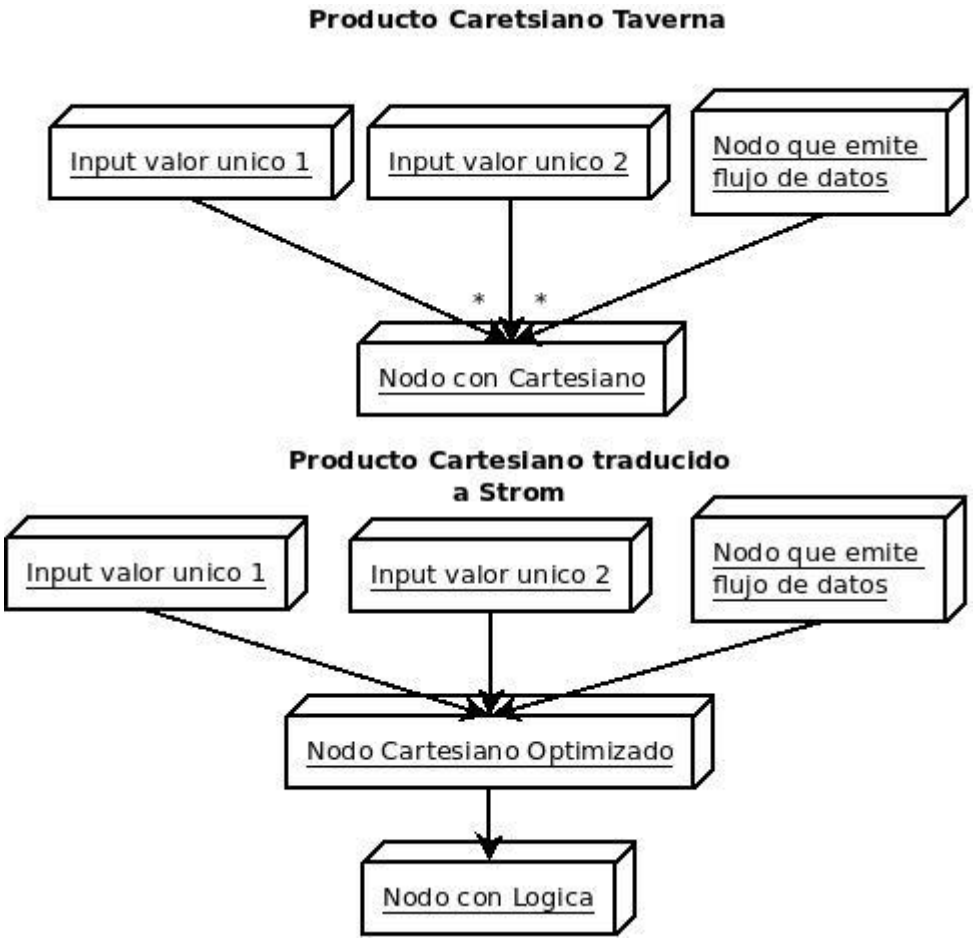

Figura 32 - Relación entre producto cartesiano definido en Taverna y estructura Storm optimizada

Esto se puede mejorar haciendo un algoritmo de ordenamiento de índices anidados de más de un nivel, con dicho algoritmo se reducen los puntos de ordenamiento a la menor cantidad posible.

En la Figura 33 se muestra un workflow con el algoritmo implementado y cómo podría ser mejorado en caso de tener un algoritmo de indexación de índices anidados de más de un nivel.

En dicha imagen queda claro que el punto de ordenamiento después de cada nodo cartesiano ya no existe y solo se produce un nodo de ordenamiento al final de la ejecución de todos los nodos cartesianos. Además de esto cabe aclarar que el ultimo nodo que refiere a un nodo que necesita un índice lineal, podría ser el nodo de salida o un nodo vectorial, únicos nodos por el momento que necesitan un índice lineal obligatoriamente.

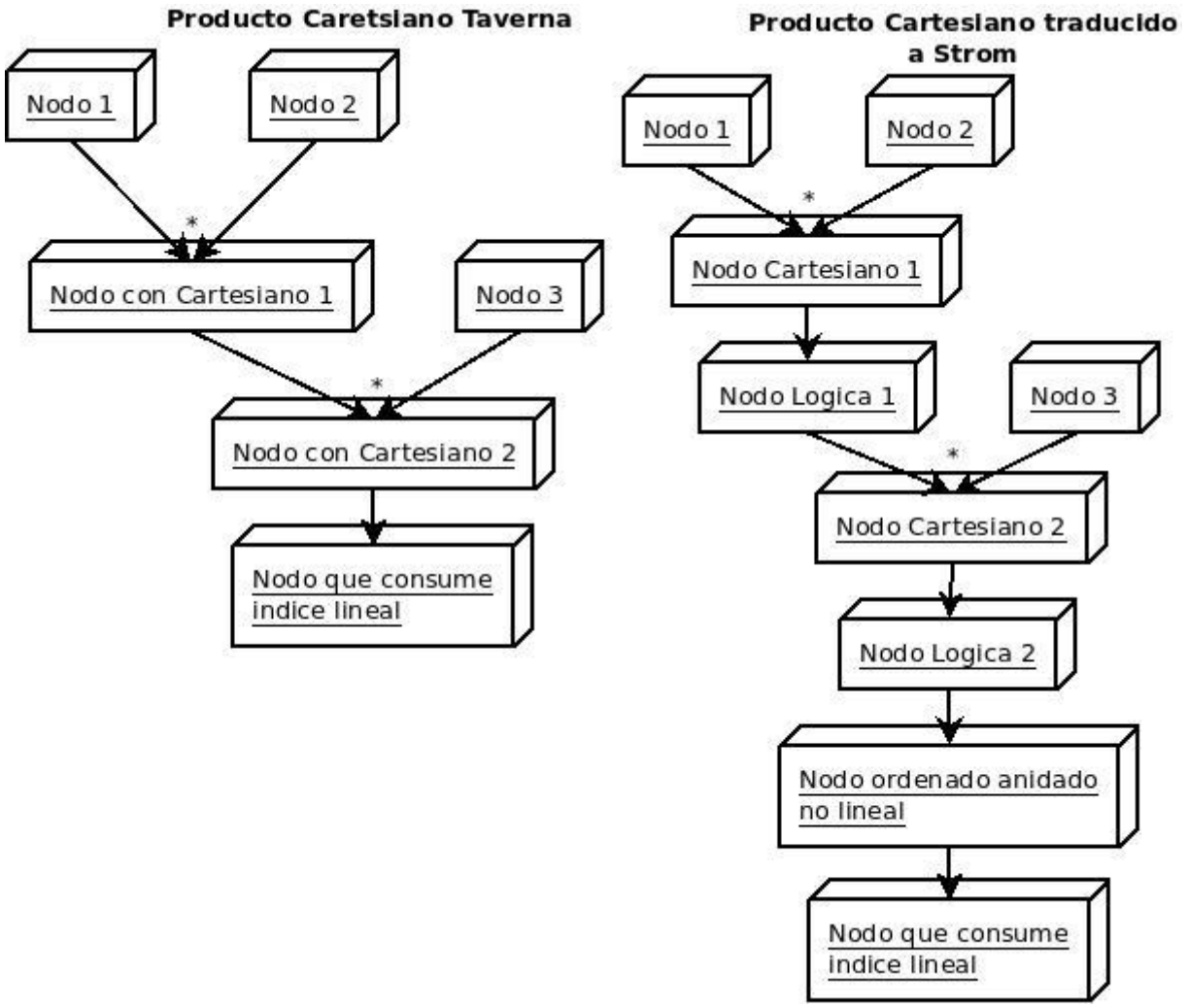

Figura 33 - Relación entre producto cartesiano definido en Taverna y estructura Storm que permite índices anidados

5.4.3 Algoritmo de finalización

Las topología en Storm nunca terminan de ejecutar como se puede ver en Definiendo procesamiento en Storm, por lo que los nodos tampoco terminan. Alguno de los procesos vistos anteriormente y algunos nodos particulares de Taverna, necesitan determinar cuándo se ha recibido el último valor del flujo.

Esto se debe a que los workflows científicos, si bien son orientados a flujos de datos, parten de un flujo de datos fijos (al menos en Taverna, aunque podrían tener una fuente “infinita” en algún momento).

Debido a esto se debió implementar un algoritmo donde cada nodo pudiera saber si estaba recibiendo la última tupla de un flujo de datos. Este algoritmo es particularmente necesario para

los nodos ordenadores anidados, ya que estos emiten todos los valores cuando han recibido ya todos los valores del flujo, y necesitan este algoritmo para saber cuándo eso ha sucedido.

Para saber más del algoritmo de finalización ver el anexo Implementación de algoritmo de finalización.

5.4.4 Guardado de provenance

Para el provenance, se agregará una capa a cada nodo desarrollado. De esta manera antes y después de procesar una tupla, se ejecutará el guardado del provenance.

Con este diseño se resuelve el provenance para todos los bolts del sistema. Así que al momento de agregar una nueva funcionalidad al sistema no tenemos que preocuparnos de gastar tiempo en el desarrollo del provenance de esta funcionalidad, ya estará solucionado por el diseño de la solución.

Para profundizar sobre la arquitectura de capas de los nodos, ver el capítulo Implementación.

5.5 Atributos de calidad

El proyecto se centró principalmente en la mejora de la performance y escalabilidad de los SWfMS, ya que en herramientas que trabajan con workflows científicos demoran mucho en ejecutarse por la gran cantidad de datos que manejan.

A continuación se hablará sobre cuatro atributos de calidad en los que se focalizó el proyecto, performance, disponibilidad, extensión y escalabilidad.

Teniendo en cuenta esto, se realizó un diseño en el que lo usuarios pueden usar la versión de Taverna Workbench para diseñar el workflow y luego poder ejecutarlo de la misma manera que lo hacían en el Server Taverna ya que las APIs serán idénticas a nivel de interfaz.

5.5.1 Performance

Para implementar una solución performante se consideraron tres características centrales que se entienden mejoran la performance total del sistema. Estas son ejecución concurrente, distribuida y tiempos bajos para la escritura y lectura de datos.

Para cumplir con ejecución concurrente y distribuida se utilizó Storm el cual implementa ambas características como se vio en la sección Aplicando Storm sobre los requerimientos.

En cuanto al acceso a datos se utilizó Redis ya que se lo consideró eficaz para las ráfagas de datos que se generarán en la solución propuesta (ver ¿Qué es Redis? en implementación).

5.5.2 Disponibilidad

Con la elección de Storm se ganó alta disponibilidad y tolerancia a fallos, ya que al momento de un fallo en una instancia de Storm siendo ejecutada en el cluster, el trabajo que estaba realizando esa instancia es asignado a otra.

Los daemons Nimbus y Supervisors en Storm son fail fast ya que la tarea se autodestruye al momento de encontrarse con alguna situación inesperada, y no guardan su estado ya que estos son mantenidos por el Zookeeper. En el caso de falle algún nodo, puedan iniciar nuevamente sin que nada hubiera pasado gracias a el Zookeeper como se puede ver en el anexo Definiendo procesamiento en Storm.

5.5.3 Extensión

Al momento de analizar las herramientas de workflows científicos, en particular Taverna, se descubrió que tenía muchas funcionalidades o servicios para realizar tareas dentro del workflow. Teniendo esto presente, se buscó que en el diseño fuera simple la forma de agregar nuevas funcionalidades a la herramienta.

El agregado de nuevas funcionalidades al sistema se logra mediante el desarrollo de una nueva clase en Java que hereda de la clase abstracta WorkberchTavernaProcessorBolt. En el capítulo Implementación se puede ver más de ella en profundidad.

5.5.4 Escalabilidad

También al elegir Storm como herramienta se tiene la posibilidad de ampliar el cluster, obteniendo así la posibilidad de escalar horizontalmente. Se logra aumentar el procesamiento al agregar más maquinas con Storm y no se necesita aumentar el poder de procesamientos de las máquinas.

6 Implementación

A lo largo de este capítulo se explicarán algunos aspectos de implementación del prototipo que se desarrolló para la solución propuesta. Se describirá el funcionamiento y el porqué de algunas tecnologías elegidas, parámetros de configuración de la solución, consumo de APIs y decisiones de implementación definidas.

6.1 Tecnologías

El prototipo cuenta con tres grandes tecnologías, el servidor web que publica la API de Taverna implementada desarrollado con Play Framework [61] para recibir y procesar los workflows a ser ejecutados en la plataforma. También cuenta con una base de datos no relacional llamada Redis [62] que almacena el provenance de los workflows y algunas variables de estado de la ejecución. Por ultimo también se utilizó la herramienta de procesamiento distribuido Storm [40] sobre el que se corren las topologías Taverna.

A continuación veremos que son y porqué se utilizaron estas herramientas, saltándonos la parte correspondiente a Storm, ya que Storm fue analizado previamente en la sección ¿Qué es Storm?

6.1.1 ¿Qué es Play Framework?

Play es un framework para Java o Scala utilizado en el desarrollo de aplicaciones web de alta productividad que implementa el modelo MVC. Está constituido por componentes y APIs necesarias para la programación de páginas web modernas.

Está basado en una arquitectura ligera, sin estado, amigable y cuenta con un consumo de recursos predecible y mínimo (CPU, memoria, hilos) para aplicaciones altamente escalables gracias a su modelo de programación reactiva, basado en Iteratee IO [63].

La arquitectura de Play está construida para programar asincrónicamente, asume que toda solicitud es potencialmente muy larga y también permite un método para programar y correr tareas de larga duración denominadas schedulers [64].

6.1.1.1 Iteratee

Iteratee provee un paradigma y una API que permite la manipulación y procesamiento de streams progresivamente con estas características: [63]

- Permitir al usuario crear, consumir y transformar streams de información.

- Tratar diferentes fuentes de datos de la misma manera (Files on disk, Websockets, Data upload, etc).

- Utilizar un amplio conjunto de adaptadores y transformadores para cambiar la forma de la fuente o el consumidor; construir propias fuentes o comenzar con primitivas.

- Tener control sobre cuándo determinar si se han enviado los datos suficientes, y ser informados cuando la fuente que los envía ha terminado de hacerlo.

- No bloqueante, reactiva y permitiendo el control sobre el consumo de recursos (Hilos, memoria).

¿Qué es Redis?

Redis es un motor de base de datos no relacional clave-valor. Se lo describe muy a menudo como un *data structure server* (servidor de datos estructurados) porque soporta trabajar nativamente con strings, hashes, listas, conjuntos, conjuntos ordenados, bitmaps y hyperloglogs [65]. También provee una forma sencilla y familiar de manipular estos tipos. [66]

6.1.1.2 Atomicidad

Redis permite un conjunto importante de operaciones atómicas sobre distintos tipos de datos, lo cual la hace muy potente como base de datos de aplicaciones distribuidas [67]:

- Concatenación de strings.

- Incrementar un valor numérico de un hash.

- Operaciones push/pop sobre una lista.

- Realizar la intersección, unión y diferencia de conjuntos.

- Obtener el elemento con mejor puntuación en un conjunto ordenado. [68]

Redis trabaja con datasets en memoria por eso puede lograr las operaciones anteriores sean atómicas.

6.1.1.3 Persistencia

La persistencia se puede lograr de dos formas: una es llamada snapshotting es un modo de persistencia donde el dataset es transferido de forma asincrónica de memoria a disco cada cierto tiempo.

La otra forma alternativa es AOF, un archivo en el que solo se le puede agregar texto al final donde se van escribiendo a medida que las operaciones modifican el dataset. También se puede permitir que no se salve a disco y que solo se trabaje en memoria [69].

6.1.1.4 Replicación

Redis soporta replicación maestro-esclavo que permite a los esclavos copias exactas del maestro [70]. A continuación se presentan algunas características de la replicación:

- Asincrónica
- Un maestro puede tener múltiples esclavos
- Los esclavos se pueden interconectar para lograr una estructura de grafo
- No se bloquean los esclavos al momento de replicar

6.1.1.5 Rendimiento

La principal característica de Redis es la performance como se puede ver en los benchmarks [71] ejecuta 80000-100000 (o más) operaciones por segundo. Queda bastante claro que es una buena opción para la mayoría de las aplicaciones que requieren respuestas rápidas (como ser aplicaciones de procesamiento en tiempo real como Storm) [72].

La performance es obtenida por el diseño minimalista de Redis donde el creador cuidadosamente eligió las características que podían ser soportadas por algoritmos rápidos y en algunos casos operaciones atómicas sin locks [72].

6.1.2 ¿Porque se eligieron Play Framework y Redis?

Para la implementación de la API Taverna se eligió utilizar Play Framework (en su versión en Java), esto se debió a que da algunas características que simplifican este trabajo. Taverna implementa toda su API a través de servicios REST y Play Framework provee las herramientas necesarias para hacerlo, ya que a través de las definiciones de controladores y el modelo de routing de Play [73], se pueden implementar servicios REST de manera rápida y sencilla.

También provee una forma de distribución del framework conocida como Activator [74], la cual provee todo lo necesario para comenzar a implementar y probar rápidamente sin previa instalación requerida, salvo una JVM.

Activator es una distribución de Play Framework que provee la herramienta de construcción SBT [75] para el manejo de dependencias y las órdenes de construcción del servidor. Además Activator también nos provee múltiples templates de aplicaciones para comenzar nuestro proyecto de manera sencilla sin tener que instalar o definir la estructura del proyecto, Activator se encarga de todo esto por sí solo.

Una de las ventajas de desarrollar en Play Framework es el cambio en caliente. Al modificar un archivo de código en Play, este automáticamente cambian en el servidor, haciendo parecer Java un lenguaje del tipo interpretado y no compilado, aunque en realidad compila una pequeña parte y lo cambia directo en el servidor.

En cuanto a Redis se eligió ya que se necesitaba una base de datos que mantuviera muy buenos tiempos tanto de escritura y lectura para aplicaciones que hicieran un uso intenso de estas operaciones y además proveer ciertas operaciones atómicas.

Al utilizar Storm, Redis se vuelve una buena solución para guardar y leer datos en los nodos de las topologías generadas, ya que se necesitamos guardar la información de provenance de cada valor generado en la topología, como información de estado necesaria para los algoritmos vistos en la sección Procesos.

6.2 Workberch Topology Builder

En esta sección veremos aspectos de implementación y diseño del Workberch Topology Builder para poder cumplir con algunos de los atributos de calidad descritos en Atributos de calidad, además de ciertas decisiones producto de utilizar las tecnologías Taverna, Storm y Redis.

6.2.1 Extensibilidad: Parser, Topology Builder y Meta-modelo

Se diseñó una capa intermedia entre la especificación del workflow y la construcción de la topología Storm. En esta capa se genera un modelo de datos de la topología para luego a partir de él construir la topología Storm. En la actualidad la plataforma utiliza la especificación de Taverna para describir los workflows científicos pero sería posible utilizar otras especificaciones implementando otro Parser. El nuevo parser sería responsable de conocer la forma en que se especifica los workflows para poder leerlos y luego poder construir el modelo de datos.

Se realizó este diseño arquitectónico porque al momento de comenzar a implementar el Parser de la especificación Taverna se vio que tenía múltiples versiones de especificaciones y también

cuando se investigó la herramienta Kepler se vio que la especificación de los workflow es distinta a Taverna. No hay un estándar establecido para la descripción de workflows y cada herramienta realiza su propia especificación. Hay algunas investigaciones de interoperabilidad entre herramientas de workflows que enfrentaron el mismo problema. [76]

Herramienta	**Especificación**
Taverna	SCUFL
Kepler	MOML
Triana	BPEL (Business Process Execution Language) y lenguaje propio
P-GRADE	Ahora usa un lenguaje propio antes usaba Condor DAG
YAWL	Lenguaje propio
K-WfGrid	GWorkflowDL

Tabla 8 - Distintas especificaciones de workflows [76]

En la Figura 34 se muestran las etapas por las que pasa la traducción de un XML de Taverna, a una definición de topología en Storm.

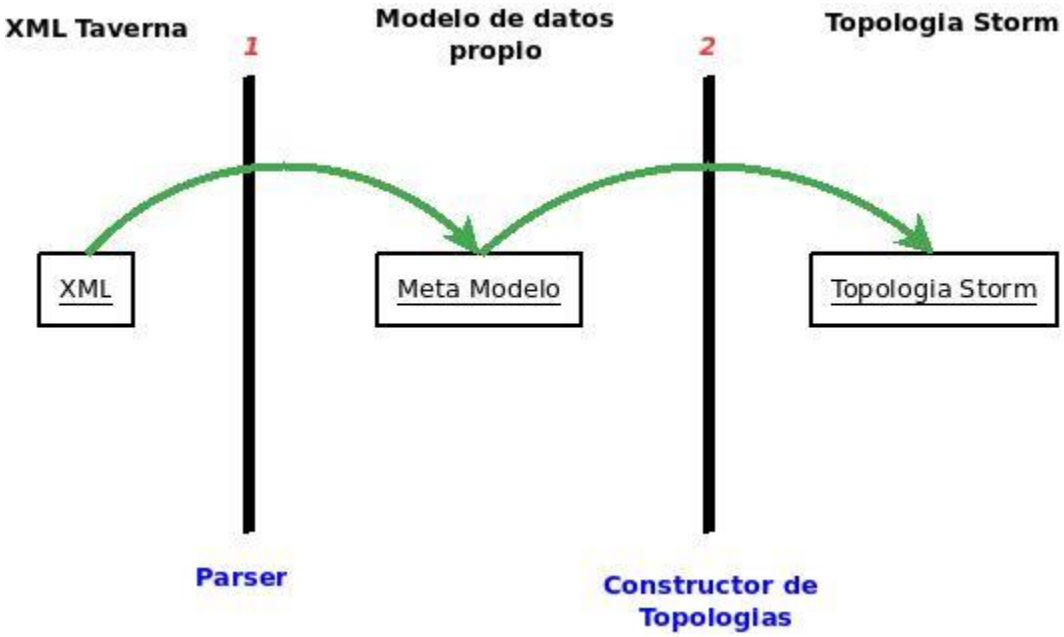

Figura 34 - Parser y constructor de topologías.

En la Figura 34 en 1, se ve la capa que transforma desde XML al modelo de datos que utiliza Workberch (Parser). En 2 se muestra como el modelo de datos es transformado a una topología Storm que es ejecutada por el cliente Storm (Constructor de Topologías).

Al construirlo de esta manera, nuestra solución permite crear un nuevo parser para cualquier generador de workflows científicos [76] o un nuevo constructor de topologías, para cualquier otro framework de trabajo distribuido como Storm.

Para la implementación de este modelo no se tomó ninguna implementación preexistente, ya que agrega complejidad al problema de Big Data que se quiere atacar en el contexto de este proyecto, pero si utilizo el concepto que se maneja en lo referenciado en [76].

A continuación vemos la definición del meta-modelo definido en el contexto de este proyecto y luego la Figura 35 donde se muestra la traducción entre los distintos modelos.

- **Logic Executer:** Representa un nodo que ejecuta lógica definida de manera programática. Se construyó el Bean Shell Logic Executer para ejecutar BeanShell (componente fundamental en Taverna). Desde el punto de vista de extensibilidad, puede haber distintos executer para distintos intérpretes.

- **Bolt Builder:** Representa un nodo que debe crear un bolt para ese tipo de nodos. Se implementaron los Output Bolt Builder (crea nodos que escriben en la salida) y Processor Bolt Builder (crea nodos intermedios en la topología) además de los nodos que ya heredan Logic Executer, que también deben heredar Bolt Builder si deben crear un bolt.

- **Data Generator:** Se mapea con elementos que produzcan datos, se implementó con Data Generator los Input Ports de Taverna y la definición de constantes de texto.

- **Node:** El homologo a Bolt Builder pero para generadores del spouts, también se usaron para los Input Ports de Taverna y constantes de texto.

- **Link:** Representa la abstracción entre una salida de un nodo, y la entrada del siguiente. Básicamente representa la traducción de una variable a otra entre diferentes nodos.

- **Iteration Strategy:** Este tipo representa las formas de interacción entre los flujos de datos. Básicamente obtiene los flujos y debe implementar un bolt en el cual el resultado sea uno o varios nuevos flujos combinados. Se utilizan para especificar el producto vectorial o el cartesiano. Esta lógica también podría ser implementada en el Bolt Builder, pero esto nos da una manera de desacoplar, en caso de ser necesario, el manejo de los flujos de su ejecución. Las implementaciones de las combinaciones se hacen a través de colocar un Iteration Strategy antes de cada Bolt Builder.

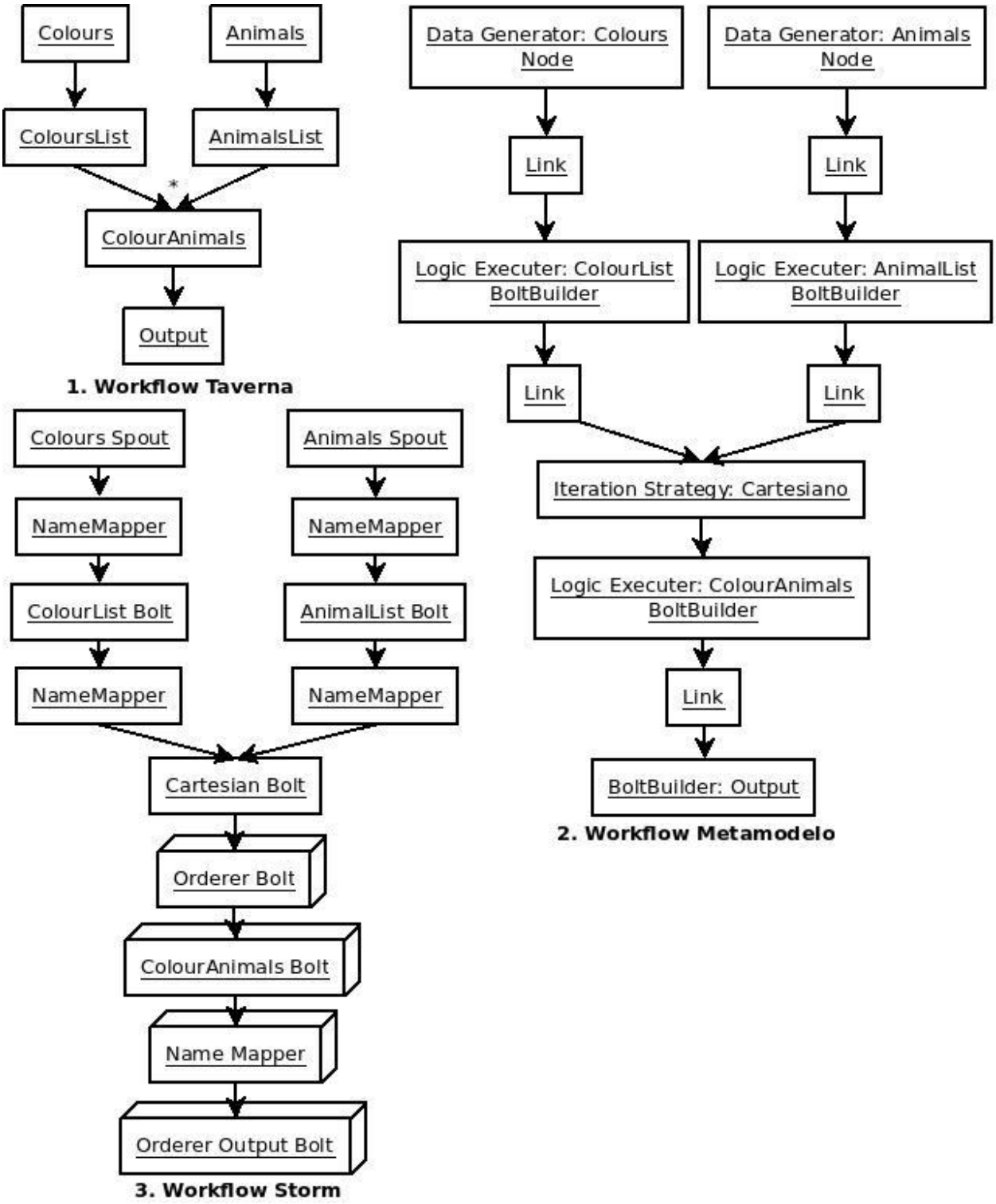

Figura 35 - En 1 un workflow Taverna, en 2 la traducción de ese workflow al meta-modelo, y en 3 el resultado en Storm.

6.2.2 Diseño de responsabilidades en capas

Storm se adaptaba bien a los workflows científicos como lo vimos en la sección Comparación de modelos: Taverna vs Storm, el uso de Storm nos condujo a tener que implementar la versión distribuida del producto cartesiano y el producto vectorial de flujos de datos.

Esto produjo un diseño en campas en cuanto a los distintos “niveles de responsabilidad” de un nodo. En la Figura 36 se puede ver las capas de responsabilidad, diseñadas para los spouts (a la izquierda) y para los bolts (a la derecha).

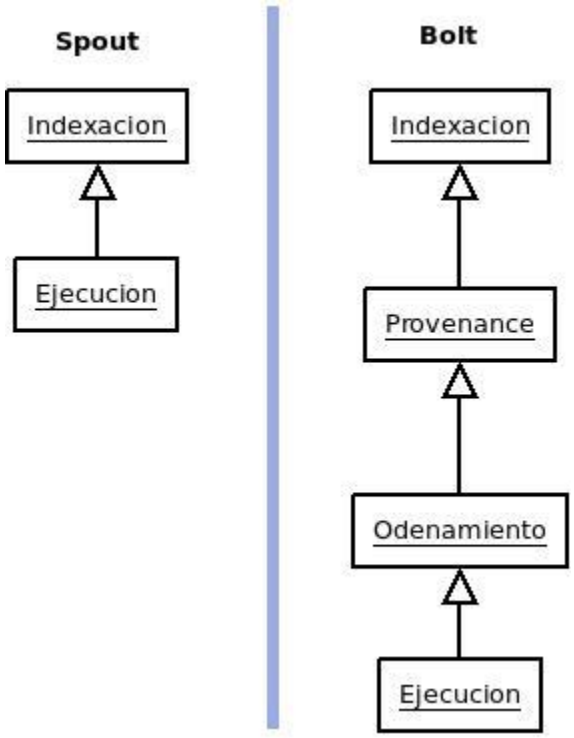

Figura 36 - Modelo de capas de responsabilidad

Cabe destacar que los nodos producidos por el meta-modelo cuando lo toma el constructor de topologías, los define en base al modelo de la Figura 36.

Veamos primero el spout de la Figura 36, el cual se divide en dos capas, estos son más sencillos ya que por el momento, todos los spouts son de una única instancia, por lo que no son multi instanciables. Los spouts solo necesitan dos capas de responsabilidad, la primera es la capa de indexación y la segunda es la de ejecución.

Los bolts en cambio en la Figura 36 implementan cuatro capas, la primera de indexación, la segunda es la provenance, una tercera de ordenamiento opcional, y la cuarta de ejecución. A continuación vemos la descripción de estas capas y para que sirve.

- **Indexación:** Lee y en el caso de los spouts produce los índices que se manejan en las capas inferiores. También en esta capa se implementa el algoritmo de finalización visto en la sección Procesos.

- **Provenance:** Esta capa se encarga de guardar toda la información de provenance, tanto de los valores recibidos como de los valores producidos.

- **Ordenamiento:** Esta capa solo la implementan los nodos que necesitan orden, al implementar esta capa, los nodos de la siguiente capa recibirán todos los nodos de forma ordenada, no importando el tipo de índice generado.

- **Ejecución:** Esta capa se encarga de hacer cualquier tipo de procesamiento, ya sea modificar índices, que es lo que hace esta capa en los nodos que implementan los productos, o ejecutar una lógica determinada para un tipo de nodo cualquiera.

6.2.3 ¿Qué nodos de Taverna son soportados y como se crean nuevos nodos?

Como se mencionó en el capítulo anterior en Extensión no se implementan todos los elementos o características que soporta Taverna pero sí se desarrollaron una cantidad mínima que permitiría realizar la mayoría de las operaciones necesarias para ejecutar un workflow científico.

Los elementos implementados son:

- **BeanShell:** Permite ejecutar scripts en Java [77]. Se busca que el usuario pueda tener la libertad de poder implementar cualquier funcionalidad que el workflow necesite.

- **XPath:** Permite navegar a través de los elementos y atributos de un documento XML.

- **REST:** Permite realizar la invocación a un servicio web REST.

- **Constantes:** Emite el valor de una constante dada.

- **Producto Cartesiano:** Combinar tuplas de datos entradas realizando el producto cartesiano de las tuplas.

- **Producto Escalar:** Combinar tuplas de datos entradas realizando el producto escalar de las tuplas.

Con estos elementos se pueden construir un conjunto importante de los workflows científicos ya que la plataforma permite la ejecución de Beanshell y se podría escribir un script Beanshell para realizar cualquier tarea particular que se necesite en el workflow. El elemento REST fue agregado porque las bases de datos científicas son accedidas, por lo general, mediante

invocaciones a servicios web y luego su información en XML es procesada mediante la ejecución de XPath.

En cuanto a la creación de nuevos componentes se puede realizar de manera muy sencilla. En caso de que se necesite un nuevo tipo de nodo, se debe extender la clase *WorkberchProvenanceBolt* la cual nos provee del servicio de provenance e indexación para cualquier tipo de nodo que necesite estos servicios, por lo que el desarrollador solo debe enfocarse en la lógica de negocio del nuevo nodo.

En caso que se quiere implementar un nodo de lógica que se inicialice a través de un XML de Taverna, se pude extender el nodo *WorkberchTavernaProcessorBolt*, el cual extiende *WorkberchProvenanceBolt,* pero además se debe implementar una función más la cual debe leer la información de la sección de XML correspondiente. Cabe aclarar que al extender de este tipo de componente, los nodos solo tienen sentido para el parser de workflows Taverna. Se decidió contar con este tipo de nodos para simplificar la creación de nodos para nuestro parser particular. La implementación *WorkberchProvenanceBolt* no depende del parser, pudiendo generar tanto nuevos nodos de control, como nodos para otro tipos de modelo de workflows que quieran ejecutar en Storm a parte de Taverna.

En caso de que el nodo a desarrollar necesite que las tuplas lleguen a la lógica de manera ordenada, se debe extender la clase *WorkberchOrderBolt*. Un ejemplo de esto es el *OutputBolt*, que es el bolt encargado de generar el archivo de salida.

6.3 Workberch Server

Se desarrolló el server utilizando el framework Play por la sencillez y robustez que brinda. La principal idea del server es que fuera sencillo agregar nuevas funcionalidades a la API REST. La simpleza de la arquitectura de Play nos da la posibilidad de poder agregarle nuevas funcionalidades a la API provista por el server de manera rápida como se vio en la sección ¿Porque se eligieron Play Framework y Redis?

6.3.1 Interacción con Cluster Storm

En esta sección se explicará cómo se realiza la comunicación entre el Workberch Server y el Cluster Storm. Principalmente hay dos tareas que son las que se encargan de comunicarse con el Cluster Storm: 1) ejecutar un workflow 2) parar la ejecución de un workflow. Las dos se realizan mediante la ejecución de una línea de comandos desde el código del server. En una primera etapa se construye el comando a ser ejecutado obteniendo los datos de configuración y en una segunda etapa se realiza la ejecución del comando construido.

Ejecutar un workflow es la tarea que cambia el status a *Operating* para que se realice el deploy de la topología en Storm. Se realiza la ejecución del comando

```
storm jar worberch-topologoy.jar main id t2flow in out runMode
```

- **storm:** Desde el lado del server se ejecuta un comando en terminal que es el cliente con el que cuenta Apache Storm, el lugar de este comando es uno de los parámetros de configuración del Workberch Server.

- **jar:** Este es un parámetro del comando storm, señala que se usará un jar para ejecutar la topología ya que este comando tiene múltiples funcionalidades con el Cluster Storm.

- **workberch-topology.jar:** Jar que contiene el generador de topologías Workberch Topology Builder, el cual es enviado al cluster para generar el workflow Storm a ejecutar.

- **main:** El main que va a ser ejecutado dentro del archivo jar enviado, en nuestro caso es el *main.java.DynamicWorkberchTopologyMain*

- **id:** Clave que identifica el workflow en el cluster.

- **t2flow:** Es el archivo con la especificación de la topología se conoce donde está por configuración en el Workberch Server.

- **in:** Es el directorio donde están los archivos de entrada para ejecutar el workflow, se conoce su ubicación por la configuración.

- **out:** Es el directorio donde se ubicaran los archivos de salida luego de la ejecución del workflow se conoce su ubicación por la configuración.

- **runMode:** Puede tener los valores *local* o *remote*. El valor *local* indica que la topología correrá en modo local (en un cluster local), mientras que *remote* indica que será en modo server. Esto se realizó para que en el proceso de desarrollo no se tuviera que tener una infraestructura con todo un Cluster Storm para poder desarrollar.

Parar la ejecución de un workflow es la tarea que para la ejecución de la topología. Se realiza la ejecución del siguiente comando:

```
storm kill id -w 30
```

- **kill:** Es el comando del cliente Storm que indica que se quiere terminar la topología con el id especificado.

- **-w 30:** También es un parámetro del comando Storm, realiza la tarea de hacer que el comando se ejecute después de 30 segundo, el Server Storm tiene un tiempo de espera por defecto para matar la topología lo que se logra con el *-w 30* es sobrescribir este tiempo.

Las partes del comando s*torm* y *id* son las mismas que en el comando anterior.

6.4 Consultas de provenance sobre Redis

El provenance guardado en Redis puede ser consultado en cualquier momento a través de Redis construyendo determinado grupo de consultas basadas en los nombre de las definiciones en la topología.

La estructura en la que se guarda el provenance se compone de HashSets [78], para cada nodo existe un hashes con toda la información sobre las tuplas recibidas, y las enviadas.

Como en Redis todo es referenciado por keys, se definió una nomenclatura de keys basada en nombre definidos en el workflow original. La key de un hash para dado un nodo, pueden ser accedidos a través de la key *[GUID].[IDNODO]* donde *GUID*, es el identificador único de la topología visto en Interacción con Cluster Storm, devuelto por el server cuando se hace deploy de un workflow. Por otro lado *IDNODO* es el nombre del nodo en el workflow definido.

Dentro del hash de un nodo se guardan tanto las tuplas recibidas, como las enviadas y cada key del hash representa un campo, de una tupla, de un nodo, ya sea recibida o enviada.

La key se expresa de la forma *[XR_|XE_][VALORINDICE].[NOMBRECAMPO].[UUID]*. Al principio se debe seleccionar *XR_ o XE_,* donde el primero es para tuplas recibidas, y el segundo para enviadas. Luego el valor de indexación de la tupla, ya sea al momento de recibir o emitir. Luego el nombre del campo que se está guardado y por último el *UUID*, este valor conecta una tupla de entrada con key que comienza con *XR_* con una o más keys que comiencen con *XE_* y represente tuplas enviadas. Con este UUID podemos determinar dada una tupla recibida, que tuplas fueron enviadas a consecuencia del procesamiento de la tupla recibida, producto de haber ejecutado la lógica.

En la Figura 37 presentada a continuación, puede verse gráficamente lo descrito con anterior en base a un ejemplo.

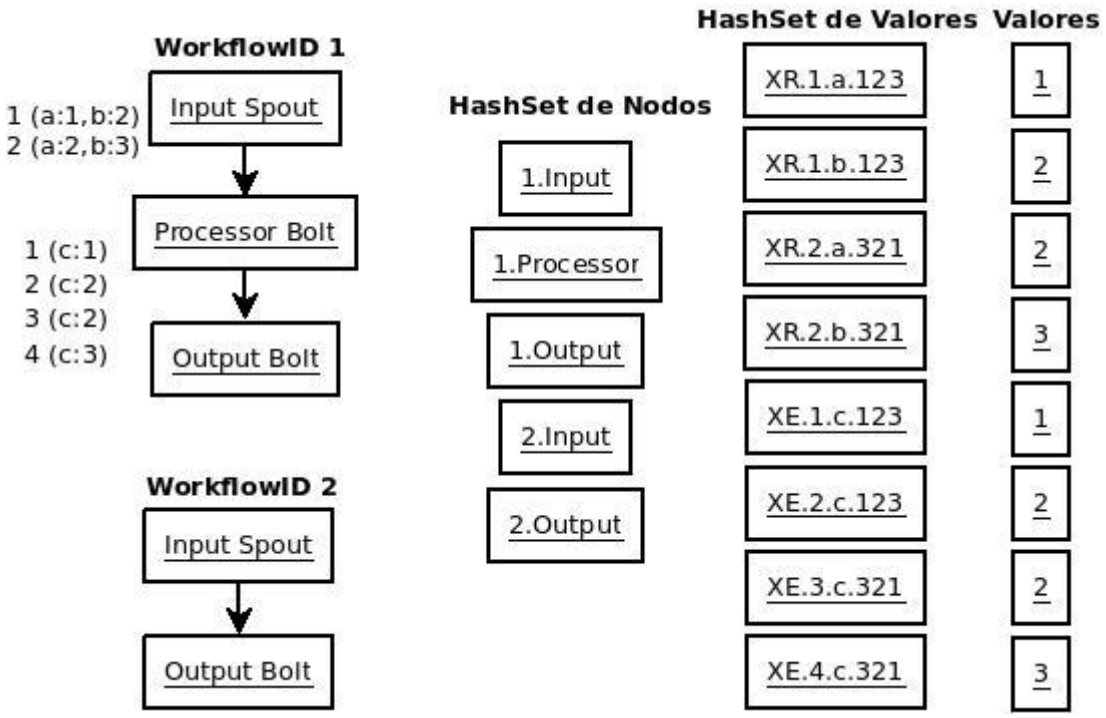

Figura 37 - Dos workflows a la izquierda, luego las keys de nodos que generan en Redis, y el hash de valores del nodo Processor con sus respectivos valores.

En este ejemplo se pueden ver dos workflows simplificados los cuales tienen ID 1 y 2 respectivamente. En el workflow de ID 1 se muestran las tuplas emitidas por el Input Spout y por el Processor Bolt para mostrar como la emisión de estas tuplas se guarda en el provenance.

El *HashSet de Nodos* representa como se guardan los nodos de todos workflows, en este caso guarda las keys de los 5 nodos de los dos workflows con las keys *[GUID].[IDNODO]*.

En el *HashSet de Valores* solo están representado el Hash del nodo Processor, pero existe uno de estos HashSets por cada key del *HashSet de Nodos*. Los valores dentro de este hash se guardan con la key *[XR_|XE_][VALORINDICE].[NOMBRECAMPO].[UUID].* En esta imagen se puede ver como son guardados los valores del workflow ID 1 y que papel juega el UUID. En el ejemplo anterior el UUID nos está diciendo que la tupla (a:1,b:2) produjo los valores (c:1) y (c:2) ya que tienen el mismo UUID.

6.5 Pruebas de Performance

Para validar el atributo de calidad visto en Performance se realizaron algunas pruebas de estrés que veremos a continuación.

6.5.1 Ejecución de pruebas

Para medir el resultado del trabajo realizado se realizaron algunas pruebas de performance enfrentando Taverna Server contra Workberch Server. Estas pruebas pueden encontrarse en anexo Tabla de resultados de pruebas de performance, también se puede encontrar en este anexo la nomenclatura de las pruebas. En estas pruebas se ejecuta Taverna Server y Workberch con diferentes configuraciones con un mismo workflow y un mismo set de pruebas.

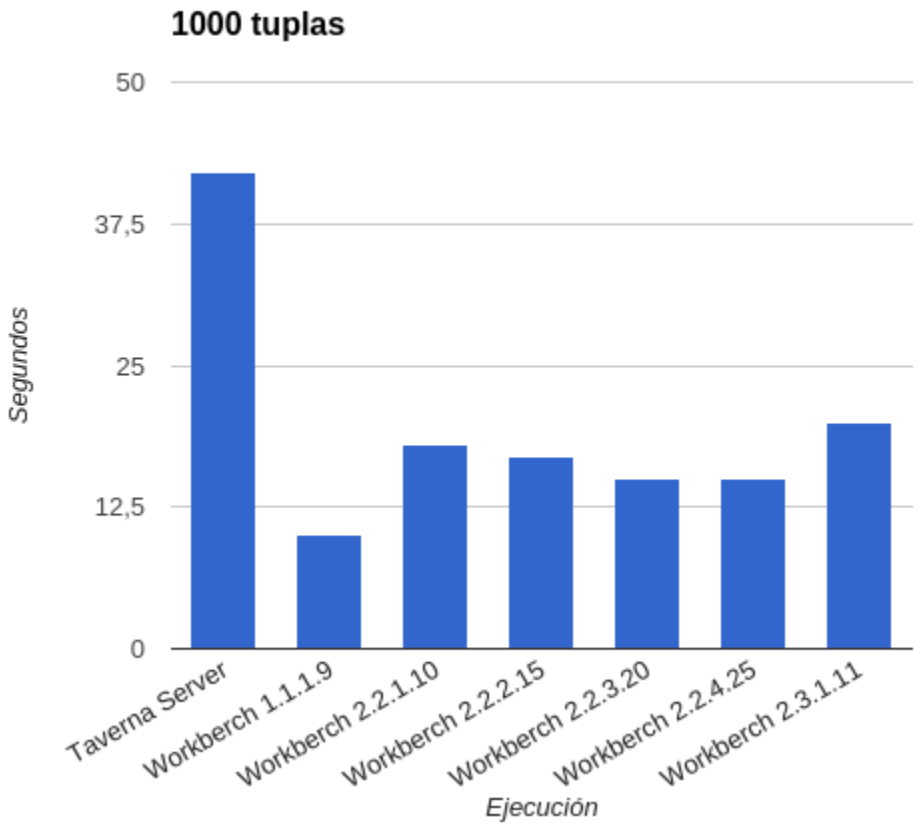

Figura 38 - Pruebas de performance para 1000 tuplas

En este primer set de pruebas de la Figura 38 se envía un volumen de 1000 tuplas a ejecutar a la topología. Como se puede ver, el caso de Taverna Server lleva mucho más que cualquiera de las ejecuciones de Workberch, además se puede ver que el caso con una sola maquina (1.1.1.9) es mucho más rápida que los casos donde se usaron varias máquinas.

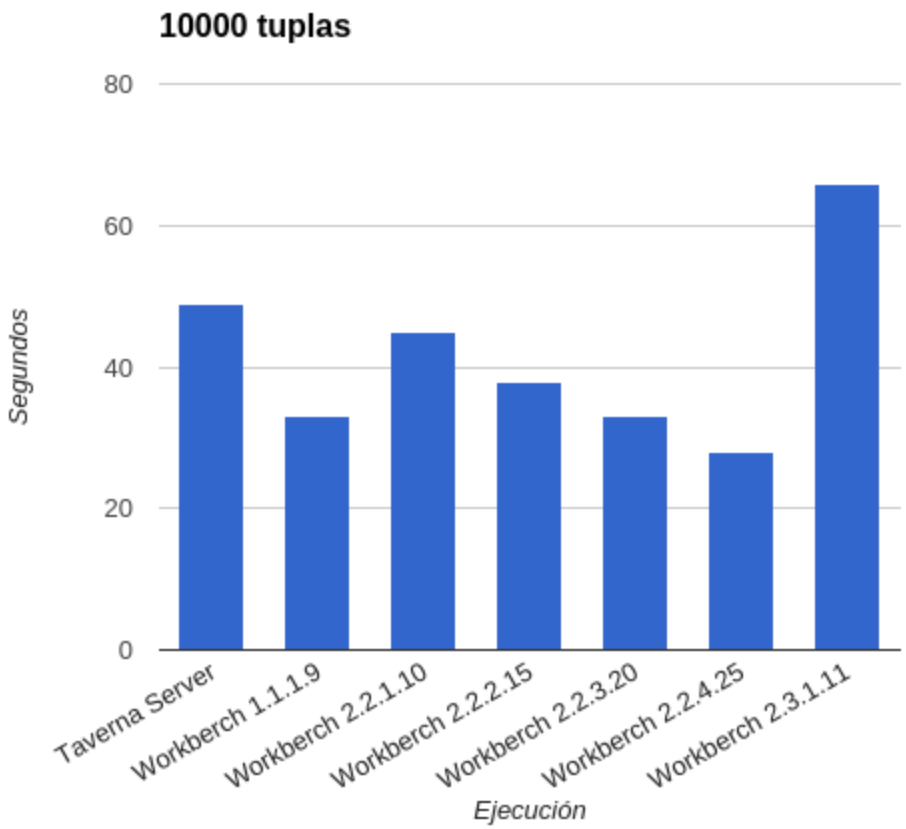

Figura 39 - Pruebas de performance para 10000 tuplas

En este segundo set de datos de la Figura 39 se envían 10000 tuplas a ejecutar a la misma topología. Para este conjunto de datos Workberch ejecuta peor que Taverna Server para el caso en el que se utilizan dos máquinas con 3 instancias de JVM (2.3.1.11).

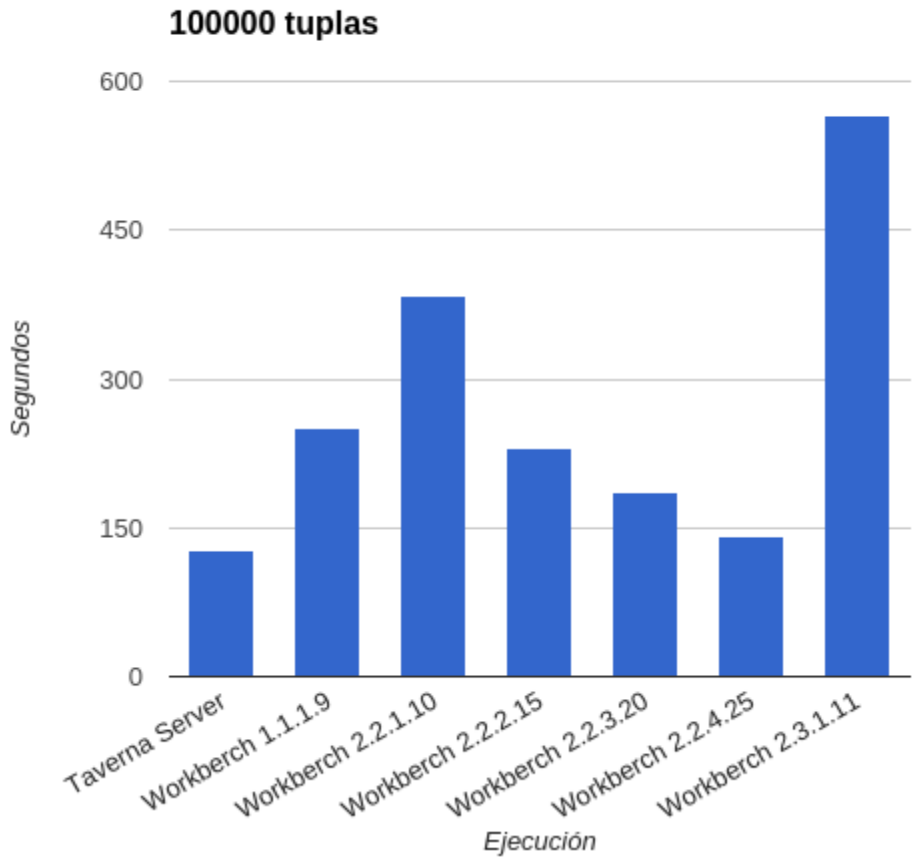

Figura 40 - Pruebas de performance para 100000 tuplas

En este tercer set de datos de la Figura 40 se envía un volumen de 100000, en este caso podemos ver que Taverna Server gana en todos los casos, por más que en el escenario de Workberch con 2 máquinas físicas, 2 JVMs y paralelización en 4 en muy similar a Taverna Server (2.2.4.25).

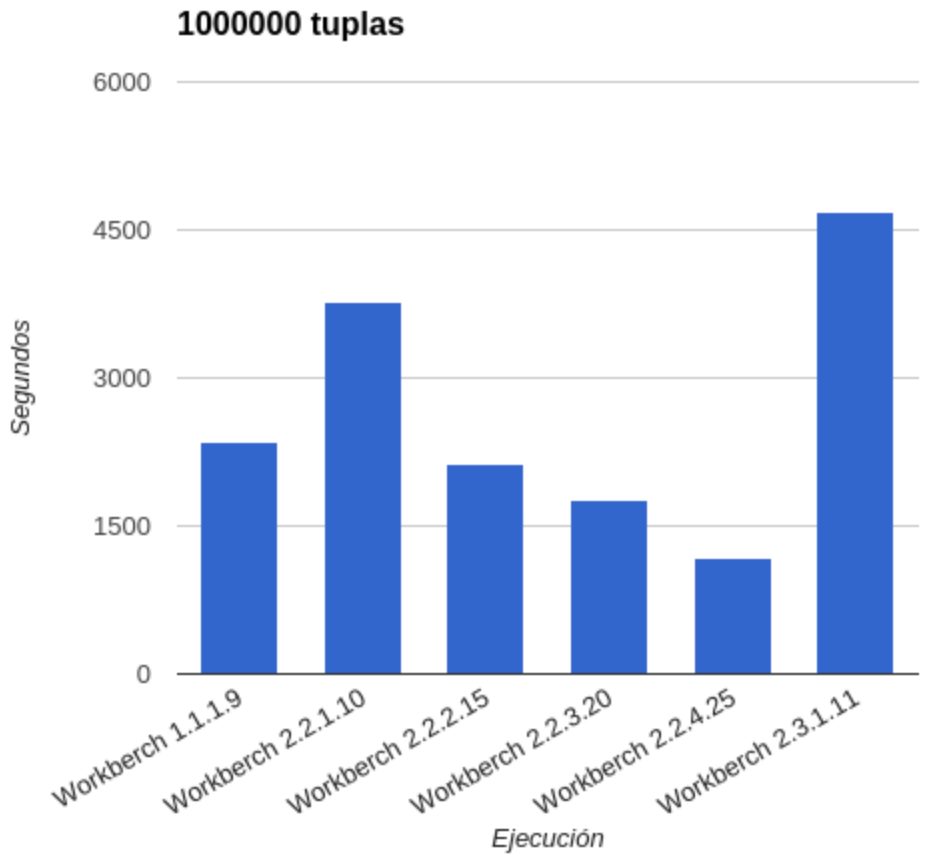

Figura 41 - Pruebas de performance para 1000000 tuplas

En este cuarto set de datos de la Figura 41 se enviaron 1000000 tuplas a ejecutar a la topología. Cabe aclarar que Taverna Server no aparece en esta ejecución porque lanza un error al tratar de ejecutar este volumen de datos. Esto se da debido a que Taverna Server escribe cada valor en un archivo por separado mientras Workberch escribe toda la salida en un solo archivo.

Antes analizar los datos se debe tener en cuenta cual era el ambiente de ejecución de pruebas con el que se contó. La infraestructura física para correr estas pruebas fueron dos máquinas físicas con cuatro cores cada una y 8GB de RAM.

En la Figura 42 vemos como se desplegaron los distintos componentes en la infraestructura de prueba para los casos 2.2. En los casos 1.1 todo corría en la misma máquina, ya que no tenía sentido la distribución de los componentes. En el caso 2.3 se colocaron dos workers en el supervisor de la máquina número 2.

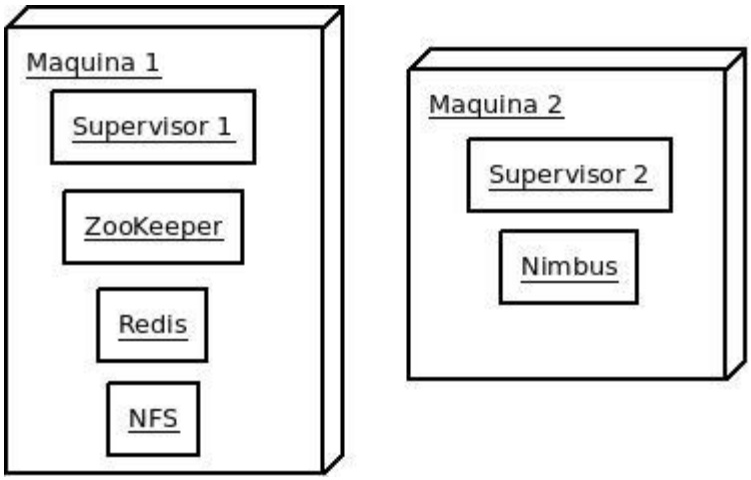

Figura 42 - Ambiente de ejecución de pruebas 2.2

6.5.2 Análisis de las pruebas realizadas

De los resultados de las pruebas se desprenden algunos resultados interesantes. Estos resultados pueden dividirse en dos categorías, los resultados de Workberch contra Workberch con diferentes configuraciones. Y los resultados Workberch contra Taverna Server.

6.5.2.1 Workberch vs Workberch

Como se puede ver en los cuatro tipos de pruebas, las pruebas 2.3 dan tiempos siempre más altos que cualquiera de las otras pruebas, con el mismo volumen de datos. Estas pruebas se ejecutaron con el objetivo de verificar si el aumento de workers en un supervisor (por ende la instanciación de dos JVMs) en una misma máquina física podía dar algún agregado de performance.

La conclusión de esta prueba es que a priori no solo no mejora la performance, sino que en general la empeora. Esto se explica debido a que los distintos threads que despliega cada worker, compiten por los recursos físicos de una misma máquina, produciendo una importante degradación en la performance.

En cuanto a las pruebas 2.2 referidas a las que usan dos máquinas con un worker cada una, dieron resultados interesantes, ya que de 2.2.1 a 2.2.4 se constató una bajada en el tiempo. Cabe aclarar que si bien se realizaron pruebas con paralelismo cinco (2.2.5), estas siempre mantenían el tiempo de la prueba 2.2.4 o lo empeoraba. De esta prueba se desprende que el paralelismo óptimo para esta distribución con dos máquinas era de cuatro.

En las pruebas 2.2 comparándola con la prueba 1.1, se logró mejorar el tiempo contra 1.1, salvo en la de volumen más pequeño (1000 tuplas de volumen). En el resto de los casos siempre se pudo mejorar la performance con respecto a la prueba 1.1. De este resultado se deduce que a mayor volumen de datos, mejor se aprovecha el paralelismo, ya que a mayor volumen se vence el tiempo del ambiente de ejecución 1.1 con menor nivel de paralelismo.

Al aumentar el volumen, el aumento del paralelismo impacta directamente en el tiempo de procesamiento, como se puede ver al comparar la prueba 1.1 con la 2.2.4 para un volumen de 1000000 de tuplas. En este caso la mejor de performance es casi del doble con respecto a la prueba 1.1 para el mismo volumen.

6.5.2.2 Workberch vs Taverna Server

En cuanto a la comparación contra Taverna Server, se puede ver que en los volúmenes de 1000 y 10000 se logra mejorar el tiempo de Taverna Server, en el caso de 100000 no se logra por muy poco en la prueba 2.2.4 y en la última no se pudieron obtener resultados debido a un error de ejecución en Taverna Server.

En una primera instancia se podría deducir que a mayor volumen, Taverna tiende a mejorar los tiempos de Workberch, aunque esto no puede asegurarse por algunos factores a tener en cuenta.

Primero debemos tener en cuenta que los tiempos de Taverna Server entre una prueba y otra no varía proporcionalmente (al aumentar diez veces el volumen no aumenta diez veces los tiempo). Esto se debe a que luego de colocar la topología en operativa, a Taverna le toma un tiempo comenzar a ejecutarla de alrededor de 24 segundos en todas las pruebas realizadas.

Workberch tiene el mismo problema, pero el tiempo que le toma comenzar a ejecutar varía mucho dependiendo del nivel de paralelización, a mayor paralelización, más tiempo le toma a Storm hacer el deploy de la topología. Aunque el tiempo en Storm es variable va entre los 10 y los 30 segundos, variando de caso a caso.

Si bien se tiene estos tiempos muertos al comienzo de la ejecución, dichos tiempo no son las demoras más relevantes de la ejecución en Workberch. Al ejecutar las pruebas se evidenció una demora realizada por una simplificación en la solución, esta demora no es menor y afecta todas las ejecuciones de Workberch Server en un tiempo proporcional al volumen de ejecución, por lo que a más volumen, mayor es la demora.

Para poder entender esta demora primero debemos entender cómo ejecuta la topología de prueba elegida de la Figura 43, donde también se muestra la traducción a topología Storm realizada por Workberch.

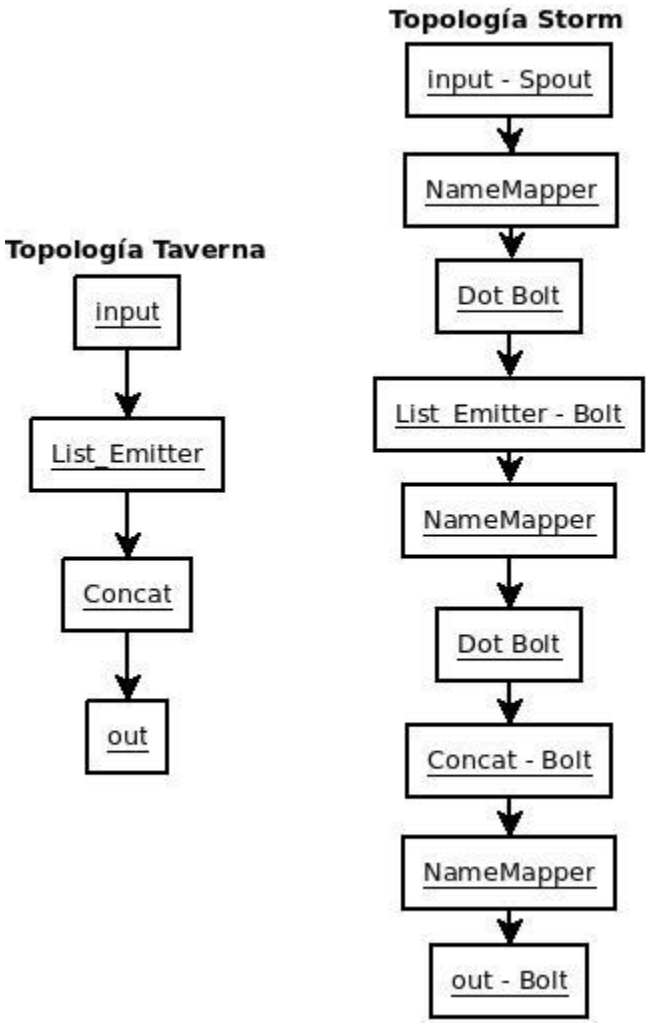

Figura 43 - Topología Taverna utilizada en las pruebas y tipología Storm generada por Workberch.

Como se ve la topología en Storm consta de ocho nodos en serie. De esta topología, solo el nodo *out - Bolt*, es un nodo ordenador (como se vio en la sección ¿Qué nodos de Taverna son soportados y como se crean nuevos nodos?

Esto significa que toda los nodos que se encuentran antes del *out - Bolt* ejecutan paralelamente. El nodo *out - Bolt* debe esperar que todos los anteriores terminen para comenzar a ejecutar, por lo que se vuelve un nodo que ejecuta de forma serial con el resto del workflow.

Este proceso serial además podría tener una demora porque el nodo *out - Bolt* es el que realiza la escritura del archivo de salida, por lo que dura en promedio más tiempo ejecutando que cualquiera de los nodos anteriores por las operaciones de IO.

Esta decisión de diseño impacto en que a mayor volumen, la demora crece proporcionalmente. Por lo que dado el tiempo de ejecución total, se observó que a el nodo *out - Bolt* ejecuta entre el 15% y el 30% de ese tiempo, luego de ejecutando todo el resto de la topología. La variabilidad del porcentaje se da porque en algunas pruebas el nodo *out - Bolt* reside en la

misma máquina el NFS y en otras no como se muestra en la Figura 43, aumentando un poco más las operaciones de IO.

Este fenómeno descrito explica por qué al aumentar los volúmenes de las pruebas en Workberch, aumentan los tiempos con respecto a Taverna. Para solucionar esto se podrían guardar los resultados en Redis o en archivos separados y no necesitar que él *out - Bolt* sea ejecutando en orden y de forma serial. Este pequeño cambio podría bajar muchísimo los tiempos de estas pruebas.

Además al aumentar el tamaño del workflow, el impacto del nodo *out - Bolt* disminuye, ya que la cantidad de resultados que escriba al final sería la misma en caso de que el workflow tuviera características similares al workflow de la Figura 43.

Por lo presentado en este capítulo se puede deducir que aunque en muchos casos Taverna ejecuta mejor que Workberch, con pequeños cambios podríamos mejorar mucho los tiempos actuales, aumentando considerablemente los casos en que Workberch sería mejor que Taverna.

6.6 Gestión de la configuración y testing

En cuanto a la gestión de la configuración se eligieron tres herramientas que facilitaron en mucho la compilación, el versionado del código y el empaquetado de los distintos artefactos del proyecto. Estos son GIT como software de control de código, Maven para el empaquetado, compilación y manejo de dependencias de Workberch Topology, y SBT para el empaquetado, compilación de Workberch Server.

Además también se desarrollaron test unitarios y de integración para algunos componentes clave del proyecto.

6.6.1 ¿Qué es GIT?

GIT es un software de manejo de control de código y versionado muy popular actualmente. A través de la utilización de este tipo de software se pudo trabajar de manera distribuida. Además GIT permite la creación de branches y tags para controlar las versiones o para mantener distintos desarrollos en paralelo.

No se utilizó ningún tipo de software “frontend” para la utilización de GIT y se optó por la utilización del cliente por línea de comandos del mismo, el cual provee toda la funcionalidad necesaria, además de ser fácil de utilizar.

6.6.2 ¿Qué es Maven?

Maven [79] es una herramienta para la gestión, construcción y manejo de dependencias de proyectos, principalmente Java, lenguaje en el cual está desarrollado. Maven es similar en funcionalidad al conocido Ant [80], aunque se le considera más simple y puede manejar dependencias remotas a través de los repositorios de Maven u otros repositorios, reduciendo la complejidad de la construcción de los artefactos.

La descripción del artefacto se hace a través de archivos XML denominados POM (Project Object Model) en el cual se define la estructura del proyecto, las dependencias y otras tareas a realizar en la construcción de los artefactos.

En el contexto de este proyecto se utilizó Maven para la construcción y gestión del artefacto Workberch Topology. Al utilizar Maven también se facilitó la integración del proyecto con los IDES incluso utilizando distintos sistemas operativos a través del plugin de Maven para Eclipse o IntelliJ. De esta manera el proyecto no guardaba ninguna dependencia a los IDES o al sistema operativo de los desarrolladores.

6.6.3 ¿Qué es SBT?

SBT [81] es otra herramienta para la gestión, construcción y manejo de dependencias de proyectos, pero a diferencia de Maven, SBT está dirigido principalmente a proyectos Scala. Además SBT permite la compilación y construcción de artefactos mixtos entre Scala y Java.

SBT maneja las dependencias utilizando Ivy [82] lo cual también le permite utilizar repositorios Maven.

SBT se utilizó en el marco de este proyecto para la gestión y construcción del artefacto Workberch Server. Se decidió utilizar SBT en Workberch Server en particular porque Play Framework viene con SBT integrado. Además todo proyecto en Play Framework debe compilar archivos Java y Scala de parte del framework, por lo que la utilización de SBT al utilizar Play es mandatoria, utilicen Java o Scala.

6.6.4 Testing

En cuanto al testing se realizaron dos tipos de pruebas, testing unitarios y testing de integración. Si bien no se llega a una cobertura particular y no hay una gran cantidad de componentes testeados se eligieron algunos puntos en particular a testear para poder hacer el desarrollo más simple en cuanto a los cambios en componentes claves del sistema.

Para las pruebas unitarias se utilizó Mockito [83] y PowerMock [84], los cuales son frameworks de pruebas unitarias con mock que combinados nos permiten crear mocks de cualquier dependencia del componente, sea pública, privada o referencia a clases estáticas.

Estas pruebas con mock se utilizaron para validar el algoritmo de indexación del producto cartesiano visto en Distribución e indexación en el producto cartesiano y el algoritmo del ordenador visto en Ordenamiento.

En cuanto a las pruebas de integración se realizaron pruebas sobre el componente que interactúa con Redis para validar que las consultas se realizarán y responden de manera correcta. Para ello se utilizó una implementación de Redis que provee un servidor embebido provista por el repositorio Maven clojars.org [85], no siendo necesario instancia un servidor Redis completo haciendo estas pruebas de mucho más sencillas.

7 Gestión del proyecto

En una primera instancia de reuniones se plantearon los requerimientos iniciales del proyecto y se comenzó una serie de investigaciones sucesivas. Estas reuniones derivaron en la caída de Hadoop como herramienta de procesamiento de workflows y la candidatura de Storm como reemplazo. Además de este cambio importante en los requerimientos iniciales del proyecto, se determinaron nuevos requerimientos producto de la elección de Storm, Taverna y Redis como herramientas centrales en el desarrollo del proyecto.

Luego de esta etapa se comenzó la etapa de diseño y desarrollo del proyecto. En esta etapa el equipo invirtió una gran cantidad de tiempo en el desarrollo de la algoritmia necesaria para la correcta ejecución de los workflows, tarea que llevó más de lo esperado por la complejidad de dichos algoritmos. Además también se hizo un gran hincapié en la extensibilidad del prototipo lo cual llevó una gran cantidad de esfuerzo en el diseño de la solución.

Siendo estos puntos los más complicados de abordar se decidió recortar el alcance y no incluir algunos puntos de los requerimientos, redefiniendo el alcance al presentado en este documento en la sección Alcance.

Luego de desarrollado el prototipo se comenzó el armado de la documentación final, la cual ya se había adelantado en la parte de investigación, generando pequeños documentos que luego serían utilizados para el desarrollo de la documentación final.

En esta fase el equipo se vio retrasado debido a problemas de enfermedad, viajes y otros contratiempos que se presentaron por parte de los integrantes del equipo.

Siendo los únicos contratiempos los mencionados anteriormente, el proyecto se desarrolló con total normalidad, tiempo y forma dentro de lo establecido.

8 Conclusiones y trabajo a futuro

8.1 Conclusiones

Hoy en día en el campo de la investigación científica, más específicamente en la experimentación in silico, se trabaja con grandes volúmenes de datos y de mucha variedad. Parte de esta experimentación es llevada a cabo a través de los SWfMS. A su vez últimamente las tecnologías de información han surgido una gran variedad de herramientas catalogadas como herramientas de Big Data, estas herramientas están pensadas para el manejo de grandes y variados volúmenes de datos. Es de interés, ver si usando las tecnologías de Big Data se puede lograr una optimización en cuanto a la performance de los SWfMS.

Al comienzo del proyecto se realizaron investigaciones de las distintas herramientas de Big Data y de Workflows Científicos. Se inició la investigación de Big Data buscando una definición de lo que es una herramienta de este tipo. Al no encontrar un consenso en cuanto a la definición de Big Data se definió una propia para usar de referencia en el marco del proyecto. Se realizó una clasificación de las distintas herramientas de Big Data en tres categorías almacenamiento, consulta y procesamiento. En cuanto a los workflows se comparó los empresariales contra los científicos, se analizaron investigaciones sobre Hadoop con Taverna y Hadoop con Kepler, además se profundizó en la investigación de los patrones multi-instancia.

En el análisis de las posibles herramientas de Big Data que podrían mejorar a los SWfMS se investigó principalmente a Hadoop pero luego de analizar otras herramientas, se concluyó que Storm es una mejor opción. También se estudiaron distintas herramientas que trabajan con SWfMS y se decidió utilizar Taverna por tener un solo modelo computacional, muchos workflows implementados públicos para ser usados, es orientado a servicios y además resultó más sencillo de utilizar que Kepler. Otra necesidad fue saber la trazabilidad de los datos dentro de los workflows (provenance), esto se realizó utilizando la base de datos Redis. Se eligió Redis porque se necesitaba una base de datos con buenos tiempos de escritura y lectura cuando es usada intensamente.

Luego de haber optado por las herramientas principales fue necesario definir mecanismos de traducción de un workflow de Taverna a topologías Storm. El equipo de trabajo decidió implementar el mismo manejo de entradas de nodos de Taverna a través de sub-topologías ya que se consideró una herramienta muy útil para la construcción de workflows. A su vez fue necesario realizar un seguimiento de los valores atraviesan un workflow para poder ordenarlos cuando fuera pertinente, esto surge de la naturaleza distribuida de Storm. También se construyeron mecanismos para la detección de finalización de un workflow dado que en Storm una topología siempre queda a la espera de más datos. Esto se debió a que el modelo de ejecución iterativo de Taverna (cartesiano y vectorial) no es de fácil implementación en Storm, siendo necesarias algunas limitaciones. La implementación del provenance tampoco fue trivial por el ordenamiento de los datos.

La solución propuesta consta de dos componentes a nivel macro: Workberch Server y Workberch Topology. Workberch Server es el punto de entrada al sistema e implementa la un subconjunto de operaciones del API definida por Taverna Server. El Workberch Topology es el componente responsable de construir las topologías Storm en base a los workflows definidos en Taverna.

Luego de implementado el prototipo, se realizaron un conjunto de pruebas de carga como validación de la realización de los atributos de calidad Performance y Escalabilidad descritos en Atributos de calidad. Ahí se pudo constatar que la solución Workberch mejora a Taverna en algunos casos en cuanto a los tiempos de ejecución (Performance), en donde se provee una solución para poder mejorar el tiempo en el resto de los casos. En el caso de escalabilidad se constata una mejora circunstancial al aumentar el hardware de la solución así como el nivel de paralelización, pudiéndose constatar la escalabilidad del sistema. Además hay que agregar que Taverna Server no funciona en cluster, por lo que la solución propuesta ya es una mejora.

Teniendo en cuenta que se logró implementar los requerimientos definidos en el alcance y que además se pudo constatar el cumplimiento de los atributos de calidad en la mayoría de los casos, se determina que todos los puntos especificados al comienzo del proyecto fueron realizados con éxito.

8.2 Trabajo a futuro

A continuación se realiza un punteo de las mejoras que se puede hacer sobre el prototipo Workberch desarrollado. En este punteo irán los requerimientos que quedaron fuera del alcance y mejoras en el diseño e implementación de lo desarrollado, ya sea porque se recortaron algunos aspectos de las funcionalidades por problemas de tiempo en el transcurso del desarrollo o porque se detectaron problemas durante la ejecución de pruebas.

- **Consumo de datos de almacenamiento masivo:** Este requerimiento (más información en Requerimientos del Proyecto) es uno de los dos requerimientos iniciales que quedaron fuera del Alcance. El desarrollo de un conector para trabajar con datos desde y hacia Redis luego de desarrollada la solución, sería un valor agregado importante, ya que podría reducir tanto los tiempos de lectura de datos, como los de escritura. Además de este conector el desarrollo de otros conectores con múltiples bases de datos NoSQL sería un agregado de valor.

- **Pausa y re-ejecución de workflows:** Este requerimiento (más información en Requerimientos del Proyecto) quedó fuera del alcance debido a que requería de una cantidad importante de trabajo y además podría realizarse luego de desarrollador el provenance, por lo que no entraba en la estimación definida originalmente. De todas maneras la realización de esta funcionalidad da un valor agregado diferencial con respecto a Taverna, ya que en la versión servidor no provee esta funcionalidad.

- **Algoritmo de índice anidado para producto cartesiano:** En la sección Ordenamiento se habló sobre el algoritmo de ordenamiento anidado y en el anexo Diseño detallado de procesos de la solución también se pueden ver algunos aspectos de este tema. En estas secciones se especifica que algoritmo de ordenamiento se terminó implementando ya que también se marca que el ordenamiento anidado fue implementado solo para anidamientos de un nivel. Sería de gran valor agregado implementar el algoritmo para más de un nivel ya que reduciría considerablemente la cantidad de puntos de ordenamiento producidos por el fenómeno visto en la Figura 31.

- **Optimización de nodos de salida:** Luego de la sección Análisis de las pruebas realizadas se identificó un problema importante en cuanto a la ejecución del nodo de salida. Para mejorar los tiempos de este nodo se propuso en esa misma sección cambiar la forma en que se escribe la salida, el cual podría ser asincrónicamente guardado sobre Redis o en distintos archivos como lo hace Taverna. Esto cambiaría radicalmente la performance de las pruebas realizadas pudiendo mejorar en mucho más casos la performance con respecto a Taverna.

- **Componentes Taverna:** En la sección ¿Qué nodos de Taverna son soportados y como se crean nuevos nodos? se vio que nodos de Taverna fueron implementados y como se pueden crear nuevos nodos. Taverna hoy en día cuenta con una gran cantidad de tipos de nodos, los cuales sería deseable implementar en caso de querer realizar un producto final de Workberch.

- **API Taverna Server:** En el anexo Taverna Server API está especificado que operaciones de la API REST de Taverna Server se implementaron. Este número de operaciones es lo que se consideró el número mínimo de operaciones que se necesitan para ejecutar un workflow cualquiera. Sería importante implementar todo el resto de la API REST de Taverna en caso de querer realizar un producto final de Workberch.

- **Ejecución en otros modelos de ejecución y especificación en otro modelo de workflows:** Para este proyecto se implementó ejecutar workflows especificados en Taverna sobre el modelo de ejecución en tiempo real de Storm. Como se vio en la sección Extensibilidad: Parser, Topology Builder y Meta-modelo se implementó un modelo en el cual sería posible integrar nuevos modelos de ejecución o especificar en otros lenguajes de workflows. Aunque el esfuerzo de este trabajo no sería menor, la integración con Workberch no implicaría una re implementación de la solución.

- **Testing:** Ahondar en los dos tipos de testing que se realizaron en este proyecto, ya sea pruebas unitarias o pruebas de performance. En el caso de las pruebas de performance sería de interés probar la solución en un ambiente a gran escala y en cluster para poder así explotar al máximo un cluster Storm. En cuanto a las pruebas unitarias serian de interés en la implementación de un producto final de Workberch para mantener la estabilidad y calidad del sistema, sobre todo en los algoritmos de alta complejidad.

9 Anexos

9.1 Teorema CAP

También se lo conoce como el teorema de Brewer, en él se afirma que solo se pueden cumplir simultáneamente con dos de las tres propiedades siguientes en un sistema distribuido [86].

- **Consistencia**: Es la propiedad C del ACID de una base de datos convencional, en todo momento los datos tiene el valor correcto sin importar en que nodo del sistema distribuido este el dato.
- **Disponibilidad (Availability)**: Solo refiere a eso, el sistema tiene que estar disponible, o no estarlo de ninguna manera. No debe haber estados intermedios de disponibilidad del sistema.

- **Tolerancia a fallos (Partition Tolerance)**: Si un nodo de nuestro sistema distribuido deja de responder, el sistema tiene que seguir funcionando a pesar del fallo.

9.2 Componentes de un cluster Storm

En capítulo de análisis se describe los conceptos de modelado de topologías en Storm, y cómo estos conceptos se interconectan para modelar el procesamiento distribuido en tiempo real. En esta sección se ve cómo Storm implementa algunos de estos conceptos, desde un punto de vista arquitectónico.

En un cluster Storm básicamente existen dos tipos de nodos, maestro o esclavo. El nodo maestro es llamado *Nimbus*, mientras que los nodos esclavos son denominados *Supervisors*.

El nodo Nimbus es el responsable de distribuir el código a ejecutar entre los distintos nodos, asignar tasks a las maquinas del cluster y monitorear fallas en las topologías.

Los nodos Supervisor reciben el código enviado por el nodo Nimbus y lo asignan a los distintos procesos asignados por Nimbus al nodo.

Toda la comunicación entre el nodo Nimbus y los Supervisor son a través de ZooKeeper [55], tanto el Nimbus como los Supervisor fail-fast [87] y sin estado, todo el estado es almacenado en ZooKeeper. La ventaja de esta arquitectura es que tanto el Nimbus como los Supervisors, pueden fallar y al volver a retomar la ejecución sin que nada hubiera pasado. De esta manera Storm provee sus atributos en cuanto a estabilidad, transaccionalidad y tolerancia a fallas.

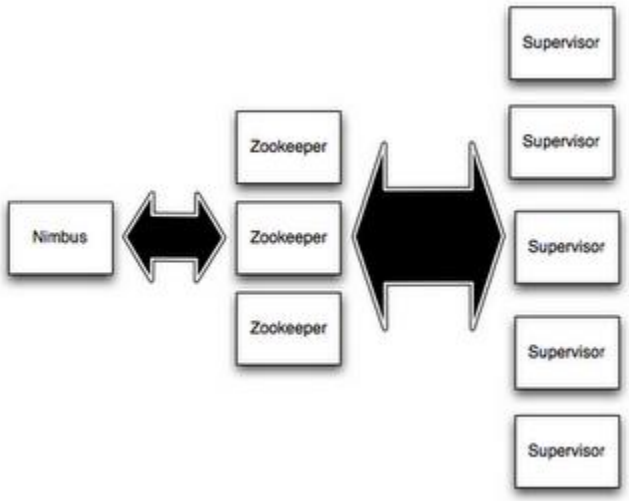

Figura 44 - Arquitectura a alto nivel de un cluster Storm [88]

En la Figura 44 se ve como la comunicación entre Nimbus y los Supervisors es a través de ZooKeeper, el cual también puede ser replicado. La replicación para ZooKeeper es parecida a la de algunos RAID, para mantener el funcionamiento de un ZooKeeper es necesario tener tres réplicas, para dos, cinco, y así sucesivamente.

9.3 Definiendo procesamiento en Storm

En esta sección veremos cómo se modela el procesamiento en Storm definiendo los principales componentes y cómo estos interactúan entre sí [89].

9.3.1 Topología

La lógica de una aplicación de procesamiento en tiempo real es modelada a través de una topología. Una topología en Storm es un grafo dirigido acíclico, donde los nodos son spouts y bolts que ejecutan la lógica y se interconecta a través de stream groupings (veremos a continuación que son estos conceptos). La particularidad de una topología es que ejecuta infinitamente (o cuando el usuario la detenga), no como un job MapReduce de Hadoop que tiene un principio y un final.

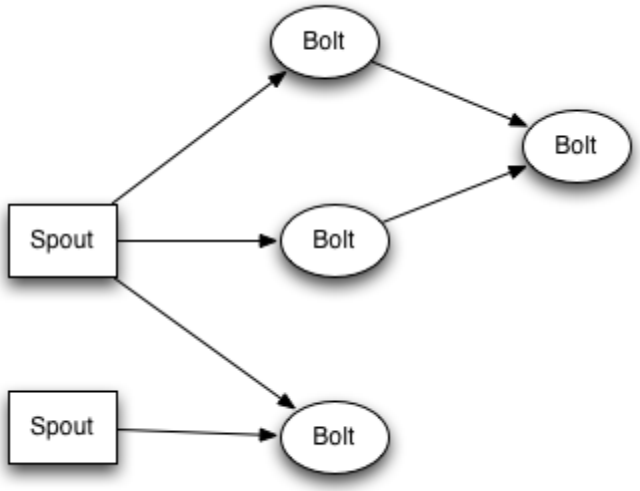

Figura 45 - Topología en Storm

9.3.2 Streams

Un stream en Storm es una serie no ordenada de tuplas que son procesadas y creadas de manera distribuida y paralela. Las tuplas son el modelo de datos que utiliza Storm para comunicarse, y pueden contener cualquier tipo de dato serializable, por lo cual es muy flexible.

Cada vez que se define un stream de tuplas, se le debe asignar un id, por lo que una topología puede manejar distintos streams a la vez. Como en general se utiliza un stream por topología, Storm cuenta con una extensa API de operaciones para no tener que proveer un identificador de stream.

9.3.3 Spouts

Los spouts definen las fuentes de datos de la topología, los spouts generalmente leerán una fuente externa de datos (aunque también podrían generales ellos mismos) y los remitirá a la topología.

Existen dos tipos de spouts, los seguros y los inseguros. Los spouts en modo seguro, re-enviaran la tupla a la topología en caso de que haya fallado el procesamiento en Storm, mientras que los spouts en modo inseguro, tan solo emitirán la tupla hacia la topología sin importarle si pudo ejecutar o no.

Los spouts son los que crean los streams, y esto lo hacen al enviar un id junto a las tuplas que emiten, determinando así que tupla va por que stream de datos.

El principal método en un spout es el método *nextTuple,* el cual será ejecutado por Storm para que el spout emita una tupla. El spout en su implementación puede enviar uno o varios valores a la topología o simplemente no enviar nada. Cabe destacar que un solo hilo ejecutará todos los spouts, por lo que no bloquear los spouts es crucial.

Otros métodos relevantes de un spout son los métodos *ack* y *fail,* estos métodos serán invocados por Storm cuando sepa que la tupla emitida terminó su procesamiento en la topología de manera satisfactoria. En caso de que la tupla falle en la ejecución, ejecutará el método fail. Estos dos métodos sólo son llamados cuando se implementa un spout seguro, no tienen sentido en un spout inseguro.

9.3.4 Bolts

Los bolts son los que contienen la lógica de procesamiento, los bolts pueden llevar a cabo cualquier procesamiento sobre las tuplas que van recibiendo, y luego pueden enviar estos resultados a otros bolts si lo requieren. Al igual que los spouts pueden manejar múltiples streams de tuplas.

Los bolts pueden hacer transformaciones simples sobre las tuplas que reciben, en caso de tener que hacer una transformación compleja, es recomendable partir este comportamiento en distintos bolts de tareas más simples, ya que esta división aumenta la posibilidad de paralelismo.

Para recibir tuplas desde un spout u otro bolt, cada bolt debe suscribirse a los bolts o spouts de los que recibe tuplas, de esta manera recibirá todas las tuplas de los elementos a los que se haya suscrito.

Los cuentan con un método *execute* el cual recibe las tuplas que llegan al bolt, luego el bolt puede emitir cero o más tuplas por cada tupla recibida, e invocar al método *ack* para confirmar los mensajes cuando sea necesario. Storm cuenta con una implementación de bolt por defecto que se puede extender para no tener que invocar *ack* cada vez, esta implementación envía un *ack* por cada tupla recibida, luego de haber ejecutado el *execute*.

Además de esto, los bolts puede si lo desean, pueden crear nuevos threads para llevar a cabo algún tipo de procesamiento asincrónico ya que los bolts son thread safe.

9.3.5 Stream grouping

Como hemos visto, los bolts se pueden suscriben a otros bolts u spouts. Estas suscripciones definen como las tuplas son distribuidas a través de la topología a través del grouping. A continuación veremos los tipos de grouping que Storm provee.

- **Shuffle grouping:** Las tuplas se distribuyen randomicamente entre las distintas instancias de los bolts suscritos de esta manera, además cada bolt recibe una cantidad igual de tuplas.

- **Fields grouping:** Agrupa las tuplas por los valores de un campo. Al definir un agrupamiento por campo, todas las tuplas con mismo valor en determinado campo de la tupla, irán hacia la misma instancia del bolt.

- **Partial Key grouping:** Un tipo particular de *Field grouping,* que es mejor para determinados casos.

- **All grouping:** Cada tupla se envía a todas las instancias del bolt suscripto.

- **Global grouping:** Las tuplas van todas a una instancia particular con determinado id.

- **None grouping:** Este grouping no define ninguna política en cuanto a cómo se distribuyen las tuplas. Aunque actualmente este se implementa igual que el *Shuffle grouping,* podría cambiar en futuras versiones.

- **Direct grouping:** En este tipo de grouping, el que envía la tupla sabe exactamente a que instancia del receptor enviarla, por lo que el mensaje es enviado a una instancia particular.

- **Local or shuffle grouping:** En este caso la tupla enviada se enviara a una instancia gestionada por el mismo worker (veremos este concepto más adelante), en caso de que no existe otra instancia en el worker, se comporta como un *Shuffle grouping.*

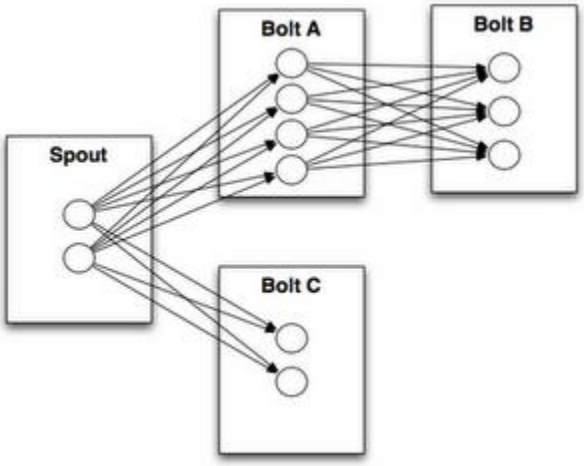

Figura 46 - Ejemplo de grouping

9.3.6 Reliability

Este concepto es implementado por Storm a través de las confirmaciones con el método *ack*. Además de esto cada mensaje cuenta con un timeout, si la tupla no termina su recorrido antes de ese tiempo, se considera perdida y se vuelve a retransmitir.

El mecanismo de cómo Storm determina que una tupla a terminado es a través del árbol de tuplas, este árbol de termina un camino dentro de la topología. En caso de que una tupla finalice este árbol, se considerar una tupla que finaliza satisfactoriamente, ya que representa que la tupla ha viajado sin problema durante todo un camino de la topología.

9.3.7 Tasks

Las task son las réplicas de los bolts y los spouts a lo largo del cluster Storm. Cada task representa un thread de ejecución, y los *Stream groupings* son los que definen como las tuplas se reparten entre las distintas tasks asociadas a un bolt o spout.

La cantidad de tasks asociadas a un bolt o spout queda determinado por el nivel de paralelismo con el que se crea al momento de desplegar la topología en Storm

9.3.8 Workers

Las topologías corren en uno o más workers donde cada worker representa una JVM en el cluster Storm. Cada worker gestiona un subconjunto de las tasks del cluster y Storm intentara distribuir las tasks a través de todos los workers para no dejar workers ociosos.

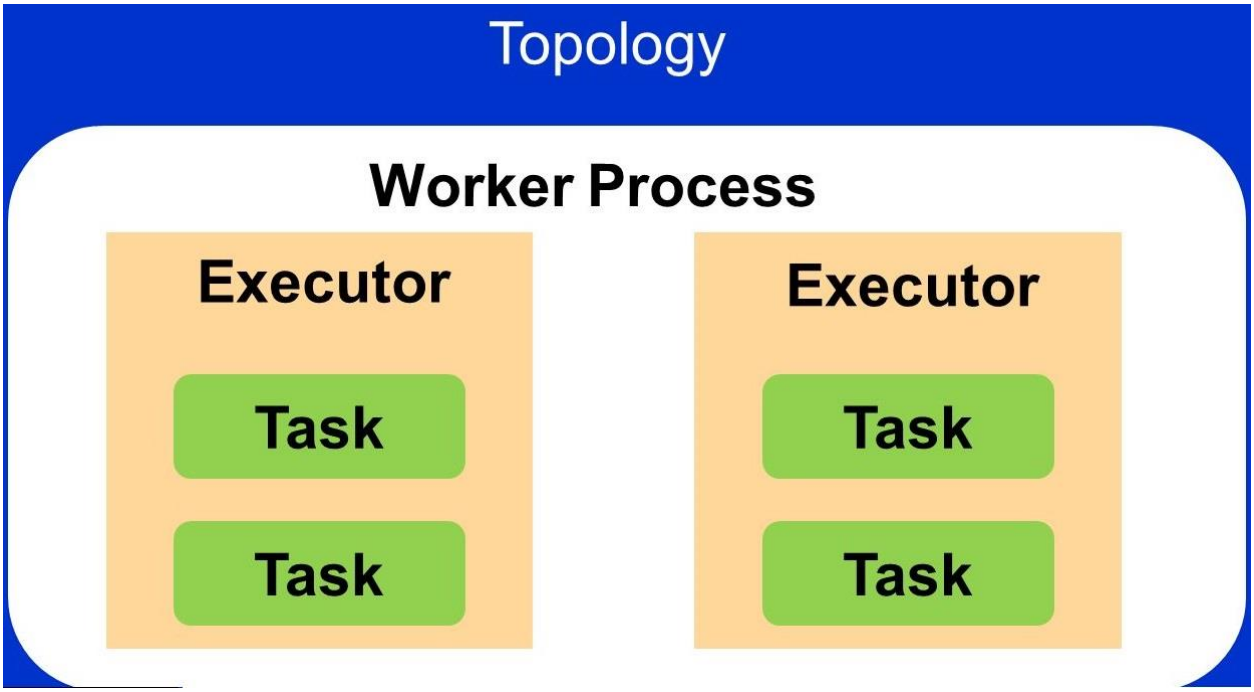

Figura 47 - Relación entre Workers y Tasks

9.4 Aplicando Storm sobre los requerimientos

La herramienta Apache Storm provee un modelo de computación distribuido en tiempo real, orientado a flujo de datos. Cómo ya se ha mencionado tiene varias similitudes con los SWfMS. A continuación se muestra cómo los conceptos de Storm se relacionan con los conceptos de workflows científicos.

- **Topología, Bolts y Spouts:** En este contexto, un workflow se mapea con una topología, los parámetros de nodos de entrada se mapean con spouts y los nodos intermedios y de salida con bolts.

- **Stream:** Esto es lo que hace a Storm una plataforma de ejecución adecuada para los workflows científicos, ya que el modelo de ejecución orientado a flujos de datos es esencial. En la Figura 48 se muestra una topología Storm con los conceptos vistos hasta ahora. Los streams son flujos de datos que pasan a través de las colecciones declaradas por el *Stream Grouping*. En la imagen los flujos pasan a través de las conexiones ilustradas como flechas en el sentido indicado.
-

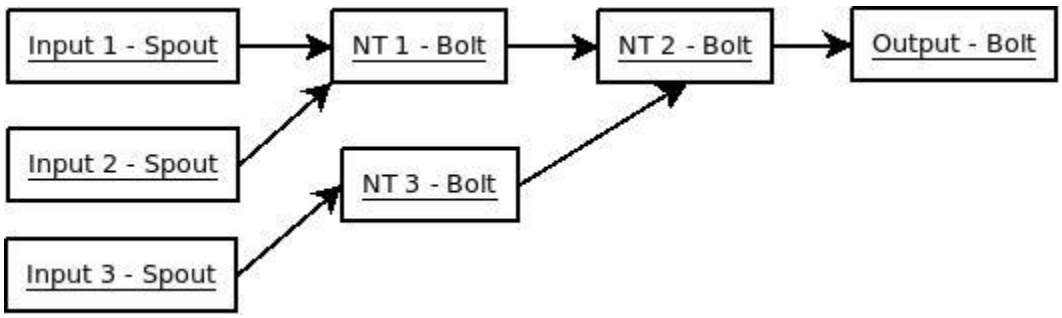

Figura 48 - Topología Storm básica

- **Tasks, executors:** Cada bolt de una topología se va mapear con un conjunto de tasks, donde cada task ejecuta exactamente la misma lógica del bolt. Cada una de estas tasks representa una instancia de ejecución y cada executor un hilo de ejecución, donde el número de task es mayor o igual a número de executors (para simplificar la solución siempre se utilizará el mismo número de tasks que de executors). Esta cantidad de hilos es el mismo concepto de paralelización que se vio en la sección de multi instancia.

- **Stream Grouping:** Apache Storm provee la funcionalidad de agrupar streams, por defecto la agrupación utilizada es “shuffle grouping”. El uso de este atributo de los streams, define la forma en que las tuplas se reparten entre las task de un bolt, con este concepto se definen las interconexiones entre los nodos.

- **Workers:** Los workers serán los encargados de gestionar los nodos y la paralelización dentro del cluster, en la Figura 49 se ven todos los conceptos agrupados en una sola imagen.

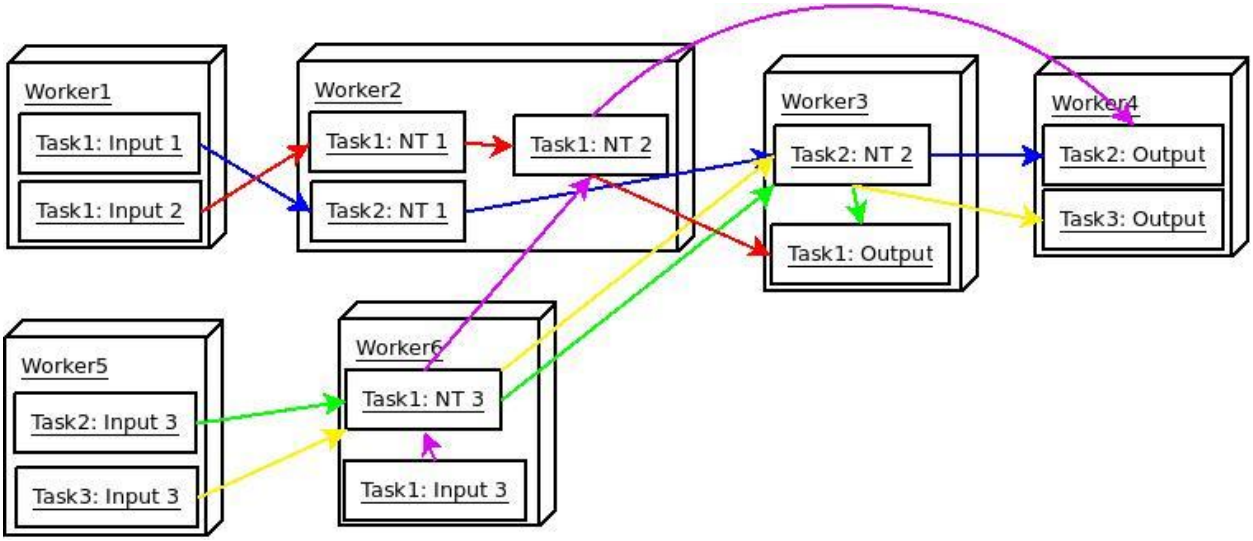

Figura 49 - Topología Storm representada con bolts, spouts y como se distribuyen entre workers y tasks

9.5 Taverna Server API

Como se mencionó en el capítulo Workberch Server el server realiza la misma API que el Taverna para que fuera sencillo a los usuarios del server Taverna usar nuestra implementación. A continuación se describe la implementación de las distintas operaciones que el server realiza.

9.5.1 Subir un workflow

Operación	HTTP	Descripción
/runs	POST	Envía un workflow al server para ser ejecutado

Cuando es enviado un workflow al server se genera una clave única que lo identifica y es almacenado el archivo recibido en una carpeta identificada con esa clave. De igual forma que el server Taverna, la clave generada es enviada en los Headers de la respuesta. Para que el usuario pueda saber como se identificara al workflow en el server.

9.5.2 Subir datos de entrada para la ejecución del workflow

Operación	HTTP	Descripción
/runs/:id/:idParam	PUT	Envía un archivo con los datos de entrada para ejecutar el workflow id.

Se salva el archivo que es enviado dentro de la carpeta identificada por la clave id según la Configuración que tenga el server con el nombre idParm.

9.5.3 Ejecutar un workflow

Operación	HTTP	Descripción
/runs/:id/status	PUT	Cambia el status del workflow id. Es usado para iniciar la ejecución

Al momento de enviar el status igual a Operating se ejecutará el jar WorkberchTopology con los parámetros establecidos según la Configuración establecida.

9.5.4 Parar la ejecución de un workflow

Operación	HTTP	Descripción
/runs/:id	DELETE	El workflow id para su ejecución

Se consulta en Redis por el estatus del workflow con clave id, si el estatus es Operating se le envía una señal a Storm para que termine la ejecución del workflow.

9.5.5 Consultar el estatus de un workflow

Operación	HTTP	Descripción
/runs/:id/status	GET	Retorna el status del workflow id

Con la clave id se busca en Redis el status del workflow.

9.5.6 Obtener los nombres de los archivos de salida

Operación	HTTP	Descripción
/runs/:id/out	GET	Retorna una lista con los nombres de los archivos de salida del workflow id

Se lee la carpeta de salida del workflow de clave id (Parámetros de configuración) y se retornan todos los nombres de los archivos que contenga.

9.5.7 Obtener los datos de un archivo de salida

Operación	HTTP	Descripción
/runs/:id/out/:part	GET	Retorna el archivo de salida part del workflow id

Se lee de la carpeta de salida del workflow de clave id (Parámetros de configuración) el archivo de nombre part y se retorna el contenido del mismo.

9.5.8 Obtener los ids de los workflows corriendo en el server

Operación	HTTP	Descripción
/runs	GET	Obtiene los ids de los workflows corriendo en el server

Se consulta en Redis por los ids de todos los Workflows que estén con el estatus Operating y estos ids son retornados.

9.6 Configuración

El server tiene algunas variables que pueden ser configuradas para poder simplificar el desarrollo y la instalación en distintas máquinas. La configuración puede ser cambiada mediante la modificación del archivo config.properties que se encuentra en el directorio raíz del Workberch Server.

topology.workflow={path-topology-folder}/guid/
topology.input.workflow={path-topology-folder}/guid/
topology.ouput.workflow={path-topology-folder}/guid/
topology.input.workflow.folder.name=in/
topology.ouput.workflow.folder.name=out/
topology.jar.file={topology-builder-folder}/{workberch-topology}.jar
storm.command={path-to-storm}/bin/storm
redis.server={redisserverip}

Tabla 9 - Un archivo de configuración de ejemplo

En la Tabla 9 se muestra la configuración donde el {path-topology-folder} es un directorio dentro de un sistema de archivos distribuidos en el cluster o carpeta compartida para los archivos de workflow. Cabe destacar la importancia de que en las rutas donde participa el {path-topology-folder} siempre debe concatenarse una carpeta guid en algún punto del path. Esto se debe a que el constructor de topologías, busca esa carpeta para generar en su lugar una carpeta con el guid de la topología, creando así un directorio por topología.

El {topology-builder-folder} es un folder que tiene que contener al jar que implementa el Topology Builder como se vio en la sección Vista de componentes.

En cuanto a {path-to-storm} y {redisserverip}, son la ruta a la instalación de Storm y IP del servidor Redis.

A continuación se describirán las variables de configuración:

- **topology.workflow:** Directorio donde se guardan los archivos con la descripción de la topología.

- **topology.input.workflow:** Directorio donde se guardan los archivos de entrada necesarios para poder ejecutar la topología.

- **topology.ouput.workflow:** Directorio donde se guardan los archivos de salida luego de finalizada la ejecución de la topología.

- **topology.input.workflow.folder.name**: Nombre del directorio dentro de la carpeta topology.input.workflow donde se almacenan los archivos de entrada. Si la variable es nula, los archivos se guardan en la ruta indicada por topology.input.workflow.

- **topology.ouput.workflow.folder.name**: Nombre del directorio dentro de la carpeta topology.output.workflow donde se almacenan los archivos de salida. Si la variable es nula los archivos se guardan en la ruta indicada por topology.output.workflow

- **topology.jar.file**: Indica la ubicación del Jar que contiene al Topology Builder.

- **storm.command**: Ruta donde se encuentran los comandos de Storm.

- **redis.server**: Tiene que ser la ip o nombre del server donde se encuentra el Redis.

9.7 Tabla de resultados de pruebas de performance

Volumen de tuplas	Taverna Server	Workberch 1.1.1.9	Workberch 2.2.1.10	Workberch 2.2.2.15	Workberch 2.2.3.20	Workberch 2.2.4.25	Workberch 2.3.1.11
1000	42	10	18	17	15	15	20
10000	49	33	45	38	33	28	66
100000	128	250	383	230	185	142	566
1000000	ERROR	2361	3778	2121	1772	1172	4677

Para identificar el tipo de ejecución en Workberch se agregó un número que determina el ambiente de ejecución. El número queda representado por *m.w.p.exe* donde m es el número de máquinas físicas utilizadas, w el número de workers que se corresponde al número de instancias de JVMs, p el número de multi instanciación a nivel de tareas y exe el número total de threads que se desplegaron para la ejecución de la topología en el cluster Storm.

9.8 Diseño detallado de procesos de la solución

9.8.1 Generación de tuplas e índices, en los spouts

Los spouts de las topologías desarrolladas serán los encargados de generar los índices. Por lo que leerá los archivos de entradas de datos de forma serial, marcando cada tupla con un contador iniciando desde cero y aumentando en uno por cada nuevo valor. De esta manera, cuando un nodo recibe una tupla sabrá a qué posición del flujo pertenece.

9.8.2 Distribución e indexación en el producto vectorial

En este caso se mostrará el comportamiento de un nodo vectorial al recibir una tupla, deberá esperar por todas las tuplas del resto de los links entrantes que tengan el mismo índice.

Para implementar dicha funcionalidad en el bolt se necesita la siguiente estructura.

```
hashmap
{
        key: "INDEX",
        value: hashmap
        {
                key: "FIELD",
                value: "VALUE"
        }
}
```

En la estructura anterior se guarda una tupla por cada valor del índice que le llega al nodo vectorial. El primer nivel del mapa tiene como claves los índices, y por cada valor del índice hay otro mapa de la forma, campo/valor para poder obtener todos los valores por cada campo para la tupla.

De todas maneras se debe recordar que un bolt puede estar multiplicado en múltiples instancias, por lo que al menos las tuplas con el mismo índice deberían ir a la misma task de un bolt dado.

Para asegurar esta condición se usa la estrategia de “field grouping” entre los links de entrada al bolt del producto vectorial. Por esta razón, la implementación del parser deberá contemplar la creación de un producto vectorial para cambiar el stream grouping de “shuffle grouping” a “field grouping” (para más información sobre Stream grouping ver anexo).

Cabe destacar que el índice de las tuplas emitidas por un producto vectorial tiene el mismo índice que las tuplas de entrada.

9.8.3 Distribución e indexación en el producto cartesiano

Para el nodo cartesiano al recibir tuplas de distintos links, este debe emitir tuplas que son la combinación de todos los valores emitidos por esos links. Para entender mejor este comportamiento se recomienda ver Figura 50.

Recordar que los nodos cartesianos también pueden ser multi-instanciados, al igual que los vectoriales, por lo que se debe definir una manera por la cual no se generen tuplas repetidas, incompletas o faltantes.

Si se supone que se está determinando los links que entran a un nodo y se tiene los nombres de los nodos predecesores de un producto cartesiano. Dada esta información se toma el primer nodo predecesor y se realiza un “shuffle grouping” para ese nodo. Para el resto de los links se hace “all grouping” el cual envía la tupla a todas las tasks del nodo siguiente.

Al repartir los links de esta manera se genera exactamente la cantidad de tuplas necesarias para representar el producto cartesiano distribuido en el flujo de datos. Esto se da porque al hacer “all grouping” una task dada tendrá el valor de todos los campos en todo momento, salvo uno de los valores de la tupla el cual irá a un solo nodo. De esta manera se logra que una instancia sola sea la encargada de producir el valor de la tupla a generar.

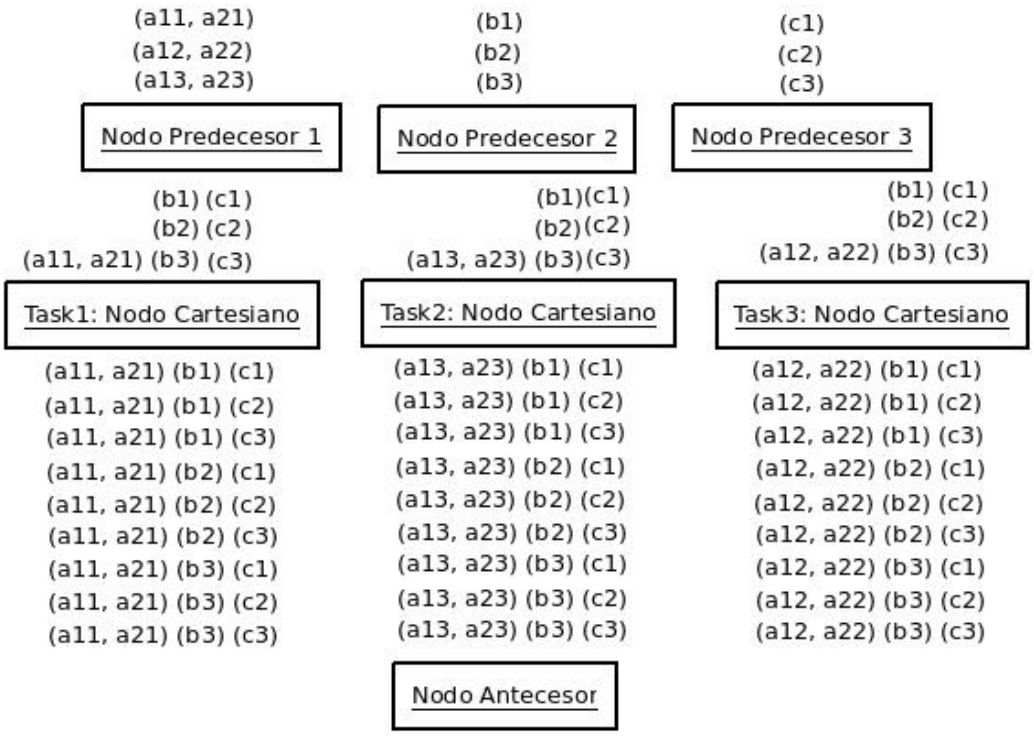

Figura 50 - Producto Cartesiano de tres entradas

En la Figura 50 se muestra como un producto cartesiano de tres entradas, provenientes de tres nodos, son recibidas por un nodo. En la parte superior se muestran los nodos predecesores y las tuplas que van a emitir.

En medio de la Figura 50 se muestra el nodo cartesiano (que se ubica entre el nodo que recibe la lógica y los que emiten las tuplas a él) multi-instanciado en tres tareas y las tuplas que recibe cada tarea con los “field grouping” definidos (las tuplas emitidas con “shuffle grouping” son emitidas aleatoriamente y las emitidas con “all grouping” son recibidas por todas las task). En la parte inferior de la imagen se muestra el nodo que recibe las tuplas ya combinadas por el producto cartesiano.

Ahora que se tiene la distribución de tuplas, cabe hacer la siguiente pregunta ¿cómo es el índice de las tuplas resultantes? El índice del cartesiano será construido como la anidación de los índices obtenidos por cada link.

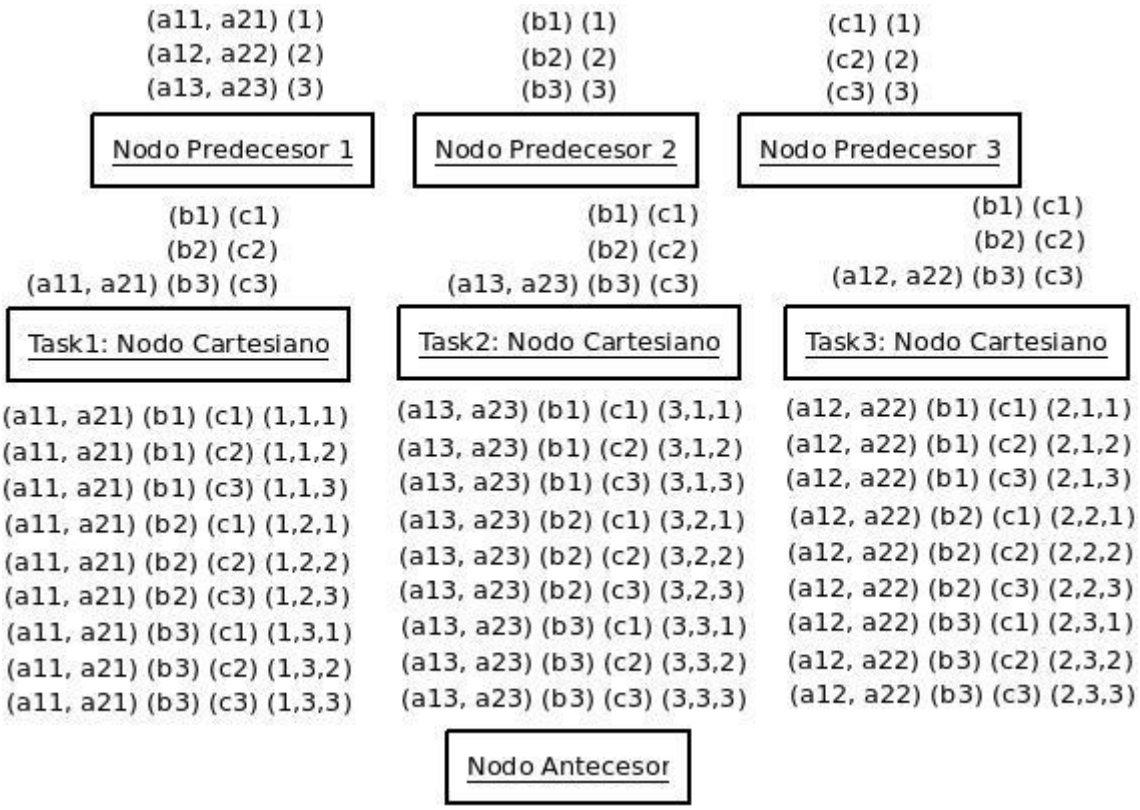

Figura 51 - Producto Cartesiano de tres entradas Indexado

De esta manera se aumenta un poco más el desorden de los índices, ya que se pasa de tener un índice lineal, a un índice anidado. Dado que para determinados puntos de ejecución se necesita orden (el ejemplo más claro es la escritura de la salida), se deben tener mecanismos para definir puntos de ordenamiento en el workflow a la hora de su generación.

9.8.4 Tipos de ordenamiento

Por lo visto en el capítulo *Indexación y distribución de tuplas* se tiene dos combinaciones posibles de entradas, y estas pueden combinarse entre sí. Estas son el producto vectorial y el producto cartesiano. Se puede apreciar que en el caso del producto vectorial, los índices recibidos deben ser todos lineales, mientras que en el caso del producto cartesiano este genera índices anidados. De esta manera se debe tener algún mecanismo para poder combinar estos dos modelos de indexación.

Este mecanismo en sí mismo es un tipo de ordenamiento, ya que sí en un punto se generó un índice anidado, se debe volver a generar un índice lineal de ese stream de datos en el caso que un producto vectorial lo reciba. Esto se da (como ejemplo) en un workflow Taverna que se ve en el siguiente diagrama.

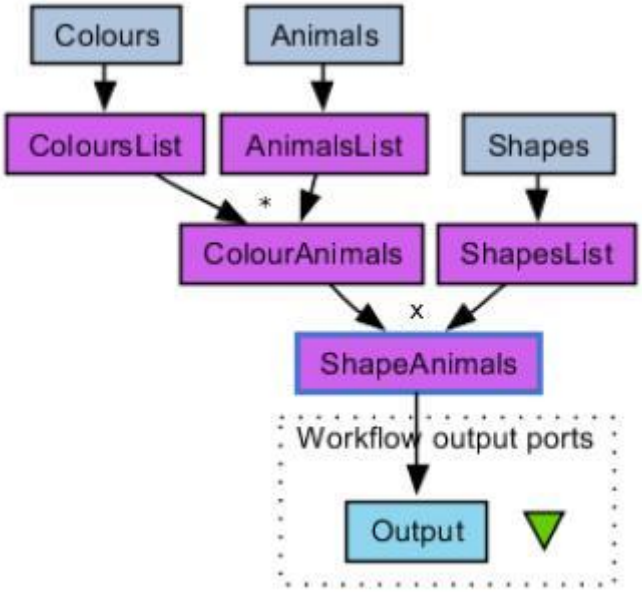

Figura 52 - Diagrama de combinación cartesiano vectorial [90]

En la Figura 52 se muestra una topología muy simple donde se muestra la combinación de los productos vectorial y escalar. ColourAnimals realiza producto escalar de las entradas ColoursList y AnimalsList denotado por el símbolo (*), mientras que ShapeAnimals realiza un producto vectorial de las entradas ColourAnimals y ShapeList denotado por el símbolo(x).

Es por esto que se definen dos tipos de ordenamiento: ordenamiento lineal y ordenamiento anidado.

9.8.5 Implementación de algoritmo de finalización

Para implementar este algoritmo se hace un pre y post procesamiento cuando se recibe una tupla y cuando se emiten nuevas tuplas. A continuación veremos lo que se hace antes y después de recibir la tupla.

```
runningNodes = list of nodes (1)

lastValue = preProcesor(tupleRecived) (2)
tupleToEmit = executeNodeLogic(lastValue)
postProcesor(tupleToEmit, lastValue) (3)

boolean preProcesor(tupleRecived) {
        prevRunningNodes = 0
        incState = incRecState(this.boltId + tupleRecived.getSourceBoltId)
        foreach node in runningNodes (4)
                if (node = tupleRecived.getSourceNode and
```

```
                    not (getFinishState(node) and incState = getEmitedState(node)))
                        prevRunningNodes++
                else if (not (getFinishState(node) and getRecState(this.boltId + node) =
                    getEmitedState(node)))
                        prevRunningNodes++    (5)
        return prevRunniongNodes = 0
}

postProcesor(tuple, lastValue) {
        incEmitedState(this.boltId)
        if (lastValue)
                setFinishState(this.boltId)
}
```

Algoritmo 3 - Implementación de algoritmo de finalización

En la línea (1) se inicializa la lista de los nombre de nodos predecesores del nodo actual en la topología, esto representa una lista de bolts o spouts y no una lista de las tasks de los mismos.

De las líneas (2) a (3) se muestra el procesamiento de cada tupla, las cuales ejecutan un pre procesamiento, luego la lógica del nodo, y luego un post procesamiento. En estas líneas también aparece la variable *lastValues* que indica si es la última tupla a emitir o no, este valor es devuelto por la ejecución de la lógica y luego utilizado también por el post procesador.

En el anterior algoritmo las funciones que modifican las variables semáforos (comunes a todas las task de un nodo) son todas atómicas, estas variables son las siguientes:

- **Received State:** Las funciones que modifica esta variable son *incRecState* y *getRecState*. Cada nodo aumenta esta variable por mensaje recibido por nodo predecesor, por lo que para cada nodo de la topología existen una cantidad N de variables de este tipo donde N es el número de nodos predecesores.

- **Emitted State:** Las funciones que las modifican son *incEmitedState* y *getEmitedState*. Esta variable existe por bolt, por lo que cada instancia del bolt incrementará y consultara esta variable. Además la variable se incrementa cada vez que se emite en una instancia, y se consulta en cualquier instancia siguiente de los bolts conectados al bolt que la incrementa.

- **Finish State:** Por cada nodo de la topología existe una sola variable de este tipo y es de tipo booleano a diferencia de las anteriores que son enteras. Las funciones para manipularla son *getFinishState* y *setFinishState*. Esta variable se inicializa en falso y se pasa a verdadero cuando un bolt emite su última tupla, cabe destacar que puede ser cualquiera de las instancias del bolt.

Notar que en cada nodo entre las líneas (4) y (5) se verifica si los nodos predecesores están aún ejecutando y si se han recibido todos los valores de los nodos predecesores. Además al

ser todo con operaciones atómicas en las variables anteriormente definidas, solo la última tupla de todos los nodos predecesores, determinara que el nodo actual coloque su estado en finalizado.

La relación entre las variables es esencial, ya que siempre se chequea primero la variable Finish State. Luego de que uno de los nodos predecesores coloque la variable Finish State en true, se pasan a chequear las cantidades recibidas por el nodo y las cantidades emitidas por los nodos predecesores. De esta manera nos aseguramos que solo una de las instancias del nodo declare el estado finalizado del nodo actual, ya que solo una obtendrá la variable local *lastValue* en true.

Además de esto, el algoritmo de finalización requiere que los spouts solo tengan una instancia, ya que tienen que saber el orden del flujo. De todas maneras esto no es un gran problema en Taverna ya que los ports de entrada y salida en Taverna no tiene sentido de paralelización ya que son una única fuente de datos ordenada.

10 Referencias

[1] «Sitio oficial de Gartner,» [En línea]. Available: http://www.gartner.com/. [Último acceso: Mayo 2014].

[2] «Perfil de Dogulas Laney en Gartner,» [En línea]. [Último acceso: Mayo 2014].

[3] D. Laney, «3D Data Management: Controlling Data Volume, Velocity, and Variety.,» META Group, 2001.

[4] W. Fan y A. Bifet, «Mining big data: current status, and forecast to the future,» 2012.

[5] S. M. Y. L. Min Chen, «Big Data: A Survey,» 2014.

[6] F. X. Diebold, «On the Origin(s) and Development of the Term 'Big Data',» 2012.

[7] «Glosario de Garnter - Big Data,» [En línea]. [Último acceso: Mayo 2014].

[8] S. Sicular, «Gartner's Big Data Definition Consists of Three Parts, Not to Be Confused with Three "V"s,» Forbes, 27 Marzo 2013. [En línea]. Available: http://www.forbes.com/sites/gartnergroup/2013/03/27/gartners-big-data-definition-consists-of-three-parts-not-to-be-confused-with-three-vs/. [Último acceso: Mayo 2014].

[9] D. Soubra, «Definicion de tres Vs Data Science Central,» [En línea]. Available: http://www.datasciencecentral.com/forum/topics/the-3vs-that-define-big-data. [Último acceso: Mayo 2014].

[10] Oracle, «Oracle: Big Data for the Enterprise,» Oracle, 2013.

[11] «Centro de noticias de Microsoft,» 11 Febrero 2012. [En línea]. Available: https://news.microsoft.com/2013/02/11/the-big-bang-how-the-big-data-explosion-is-changing-the-world/. [Último acceso: Mayo 2014].

[12] R. B. Fragoso, «IBM developerWorks - ¿Qué es Big Data?,» 18 Junio 2012. [En línea]. Available: https://www.ibm.com/developerworks/ssa/local/im/que-es-big-data/. [Último acceso: Mayo 2014].

[13] «Puro Martketing,» 2014. [En línea]. Available: http://www.puromarketing.com/12/22219/realmente-seguro-big-data-para-usuarios-consumidores.html. [Último acceso: Mayo 2014].

[14] F. Lo, «DataJobs,» [En línea]. Available: https://datajobs.com/what-is-hadoop-and-nosql. [Último acceso: Julio 2014].

[15] D. J. Power, «What is ACID and BASE in database theory?,» 10 Diciembre 2013. [En línea]. Available: http://dssresources.com/faq/index.php?action=artikel&id=281. [Último acceso: Mayo 2014].

[16] C. Roe, «ACID vs. BASE: The Shifting pH of Database Transaction Processing,» 1 Marzo 2012. [En línea]. Available: http://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/. [Último acceso: Junio 2014].

[17] W. Vogels, «All Things Distributed - Eventually Consistent - Revisited,» 22 Diciembre 2008. [En línea]. Available: http://www.allthingsdistributed.com/2008/12/eventually_consistent.html. [Último acceso: Junio 2014].

[18] M. Cooper y P. Mell, «Tackling Big Data,» 2012. [En línea]. Available:

http://csrc.nist.gov/groups/SMA/forum/documents/june2012presentations/fcsm_june2012_cooper_mell.pdf. [Último acceso: Junio 2014].

[19] «NOSQL Databases,» [En línea]. Available: http://nosql-database.org/. [Último acceso: Junio 2014].

[20] «Sitio oficial de neo4j,» [En línea]. Available: http://neo4j.com/. [Último acceso: Junio 2014].

[21] «Sitio oficial de Infinity Graph,» [En línea]. Available: http://www.objectivity.com/products/infinitegraph/. [Último acceso: Junio 2014].

[22] «Sitio oficial de Titan,» [En línea]. Available: http://thinkaurelius.github.io/titan/. [Último acceso: Junio 2014].

[23] «Sitio oficial de HBase,» [En línea]. Available: http://hbase.apache.org/. [Último acceso: Junio 2014].

[24] «Sitio oficial de Cassandra,» [En línea]. Available: http://cassandra.apache.org/. [Último acceso: Junio 2014].

[25] «Sitio oficial de Hypertable,» [En línea]. [Último acceso: Junio 2014].

[26] «Sitio oficial de MongoDB,» [En línea]. Available: https://www.mongodb.org/. [Último acceso: Junio 2014].

[27] «Sitio oficial de CouchDB,» [En línea]. Available: http://couchdb.apache.org/. [Último acceso: Junio 2014].

[28] «Sitio oficial de Clusterpoint,» [En línea]. Available: https://www.clusterpoint.com/. [Último acceso: Junio 2014].

[29] «Sitio oficial de Redis,» [En línea]. Available: http://redis.io/. [Último acceso: Junio 2014].

[30] «Sitio oficial de RiakKV,» [En línea]. Available: http://riak.basho.com/. [Último acceso: Junio 2015].

[31] «Sitio oficial de Berkeley,» [En línea]. Available: http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html.

[32] «Sitio oficial Solr,» [En línea]. Available: https://lucene.apache.org/solr/. [Último acceso: Junio 2014].

[33] «Sitio oficial de Apache Cuckwa,» [En línea]. Available: https://chukwa.apache.org/. [Último acceso: Junio 2014].

[34] «Sitio oficial de Pig,» [En línea]. Available: https://pig.apache.org/. [Último acceso: Junio 2014].

[35] «Sitio oficial de Apache Mahout,» [En línea]. Available: http://mahout.apache.org/. [Último acceso: Junio 2014].

[36] «Sitio oficial de GraphLab,» [En línea]. [Último acceso: Junio 2014].

[37] «Sitio oficial de Google Big Query,» [En línea]. Available: https://cloud.google.com/bigquery/. [Último acceso: Junio 2014].

[38] «Sitio oficial de Micrososft HDInsight,» [En línea]. Available: http://azure.microsoft.com/es-es/services/hdinsight/. [Último acceso: Junio 2014].

[39] «Sitio oficial de Apache Hadoop,» [En línea]. Available: https://hadoop.apache.org/. [Último

acceso: Junio 2014].

[40] «Sitio oficial de Apache Storm,» [En línea]. Available: https://storm.apache.org/. [Último acceso: Junio 2014].

[41] «Sitio oficial de Apache Sqoop,» [En línea]. Available: http://sqoop.apache.org/. [Último acceso: Junio 2014].

[42] «Sitio oficial de Apache Spark,» [En línea]. [Último acceso: Junio 2014].

[43] «Sitio oficial de Apache Oozie,» [En línea]. Available: https://oozie.apache.org/. [Último acceso: Junio 2014].

[44] «Sitio oficial de Apache Flume,» [En línea]. Available: http://flume.apache.org/. [Último acceso: Junio 2014].

[45] «Patente de Google Map Reduce,» 18 Junio 2014. [En línea]. Available: http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=/netahtml/PTO/srchnum.htm&r=1&f=G&l=50&s1=7,650,331.PN.&OS=PN/7,650,331&RS=PN/7,650,331. [Último acceso: Agosto 2014].

[46] «Sitio oficial de la Workflow Management Coalition,» [En línea]. Available: http://www.wfmc.org/. [Último acceso: Julio 2014].

[47] «Taverna - Why use workflows?,» [En línea]. Available: http://www.taverna.org.uk/introduction/why-use-workflows/. [Último acceso: Mayo 2014].

[48] U. Yildiz, A. Guabtniy y A. H. Nguz, «Business versus Scientific Workflow: A comparative Study,» 2009.

[49] S. Migliorini, M. Gambini, M. L. Rosa y A. H. t. Hofstede, «Pattern-Based Evaluation of Scientific Workflow Management Systems,» 2011.

[50] «Sitio oficial de Kepler,» [En línea]. Available: https://kepler-project.org/. [Último acceso: Mayo 2014].

[51] J. Wang, D. Crawl y I. Altintas, «A Framework for Distributed Data-Parallel Execution in the Kepler,» 2012.

[52] «Sitio oficial de Taverna,» [En línea]. Available: http://www.taverna.org.uk/. [Último acceso: Agosto 2014].

[53] «Taverna PBS,» [En línea]. Available: http://cphg.virginia.edu/mackey/projects/sequencing-pipelines/tavernapbs/. [Último acceso: Jlio 2014].

[54] «Storm - Rationale,» [En línea]. Available: http://storm.apache.org/documentation/Rationale.html. [Último acceso: Julio 2014].

[55] «Sitio oficial de Apache Zookeeper,» [En línea]. Available: https://zookeeper.apache.org/. [Último acceso: Junio 2014].

[56] «My Grid - Blue Version of Hadoop Parallelism,» [En línea]. Available: http://dev.mygrid.org.uk/wiki/display/scape/Blue+Version. [Último acceso: Julio 2014].

[57] «My Grid - Green Red Version of Hadoop Parallelism,» [En línea]. Available: http://dev.mygrid.org.uk/wiki/display/scape/GreenRed+Version. [Último acceso: Julio 2014].

[58] J. Wang, D. Crawl y I. Altintas, Kepler + Hadoop : A General Architecture Facilitating Data-

Intensive Applications in Scientific Workflow Systems, 2009.

[59] «Strom Rationale,» Apache, [En línea]. Available: https://storm.apache.org/documentation/Rationale.html. [Último acceso: Julio 2014].

[60] The University of Manchester, University of Southampton, «MyExperiment - Workflows,» [En línea]. Available: http://www.myexperiment.org/workflows. [Último acceso: Julio 2014].

[61] «Play Framework,» Agosto 2014. [En línea]. Available: https://www.playframework.com.

[62] «Redis,» Agosto 2014. [En línea]. Available: http://redis.io.

[63] «Play Framework Iteratees,» Agosto 2014. [En línea]. Available: https://www.playframework.com/documentation/2.0/Iteratees.

[64] «Play Framework Philosophy,» Agosto 2014. [En línea]. Available: https://www.playframework.com/documentation/2.3.x/Philosophy.

[65] «Redis Hyperloglogs,» Agosto 2014. [En línea]. Available: http://redis.io/topics/data-types-intro#hyperloglogs.

[66] «An introduction to Redis data types and abstractions,» Agosto 2014. [En línea]. Available: http://redis.io/topics/data-types-intro.

[67] «Introduction to Redis,» Agosto 2014. [En línea]. Available: http://redis.io/topics/introduction.

[68] «An introduction to Redis data types and abstractions,» Agosto 2014. [En línea]. Available: http://redis.io/topics/data-types-intro.

[69] «Redis Persistence,» Agosto 2014. [En línea]. Available: http://redis.io/topics/persistence.

[70] «Redis Replication,» Agosto 2014. [En línea]. Available: http://redis.io/topics/replication.

[71] «Redis Benchmarks,» Agosto 2014. [En línea]. Available: https://code.google.com/p/redis/wiki/Benchmarks.

[72] «Redis: Lightweight key/value Store That Goes the Extra Mile,» Agosto 2014. [En línea]. Available: http://www.linux-mag.com/id/7496/.

[73] «Play Framework - HTTP Routing,» Agosto 2014. [En línea]. Available: https://www.playframework.com/documentation/1.0.1/routes.

[74] «Play Framework - What’s new on Play 2.3,» Agosto 2014. [En línea]. Available: https://groups.google.com/forum/#!msg/play-framework/bTvJbeR_zvU/J3reqk6Xo4AJ.

[75] «Scala - SBT,» Agosto 2014. [En línea]. Available: http://www.scala-sbt.org/.

[76] T. K. G. T. Tamas Kukla, «Enabling the execution of various workflows (Kepler, Taverna, Triana, P-GRADE) on EGEE,» [En línea]. Available: http://twiki.oats.inaf.it/twiki/pub/ADCIs/ErFlowMeetingGranada/OGF25_final-wf-peter-gabor.ppt.

[77] «Beanshell,» Julio 2014. [En línea]. Available: http://www.beanshell.org/intro.html.

[78] «Hash Sets operations on Redis,» Agosto 2014. [En línea]. Available: http://redis.io/commands#hash.

[79] «Apache Maven Project,» Junio 2014. [En línea]. Available: http://maven.apache.org/.

[80] «Apache Ant,» [En línea]. Available: http://ant.apache.org/.

[81] «sbt,» [En línea]. Available: http://www.scala-sbt.org/.

[82] «Ivy - The agile dependency manager,» [En línea]. Available: http://ant.apache.org/ivy/.

[83] «Tasty mocking framework for unit tests in Java,» [En línea]. Available: http://mockito.org/.

[84] «powermock github,» [En línea]. Available: https://github.com/jayway/powermock.

[85] «Redis embedded server for Java integration testing,» [En línea]. Available: https://clojars.org/redis.embedded/embedded-redis.

[86] B. C. Theorem, 2012. [En línea]. Available: http://webpages.cs.luc.edu/~pld/353/gilbert_lynch_brewer_proof.pdf .

[87] M. Fowler, «Fail Fast principles,» Octubre 2014. [En línea]. Available: http://www.martinfowler.com/ieeeSoftware/failFast.pdf.

[88] «Apache Storm Tutorial,» Junio 2014. [En línea]. Available: http://storm.apache.org/documentation/Tutorial.html.

[89] «Apache Storm Concepts,» Junio 2014. [En línea]. Available: http://storm.apache.org/documentation/Concepts.html .

[90] «Demonstration of configurable iteration,» [En línea]. Available: http://www.myexperiment.org/workflows/822.html. [Último acceso: Agosto 2014].