Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

Doctorado en Software, Sistemas y Computación

Tesis Doctoral

Gestión de Datos Masivos: Carga y Elasticidad

Autora: Ainhoa Azqueta Alzúaz
Máster Universitario en Software y Sistemas

Directora: Marta Patiño Martínez
Doctora en Informática

Madrid, Diciembre de 2020

Esta Tesis Doctoral se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Tesis Doctoral
Doctorado en Software Sistemas y Computación

Título: Gestión de Datos Masivos: Carga y Elasticidad

Diciembre de 2020

Autor: Ainhoa Azqueta Alzúaz
Máster Universitario en Software y Sistemas

Directora: Marta Patiño Martínez
Doctorada en Informática

Lenguajes y Sistemas Informáticas e Ingeniería de Software
ETSI Informáticos
Universidad Politécnica de Madrid

Agradecimientos

En primer lugar, quiero dar las gracias a mi directora Marta Patiño, por guiarme en este camino y creer en mí. Por todas las revisiones, consultas y consejos. Desde aquel septiembre de 2013 que comencé a trabajar en el laboratorio me has enseñado lo que es trabajar duro para conseguir lo que uno desea.

También quiero dar las gracias a José Enrique Armendariz. Gracias por ficharme a finales del 2010 y enseñarme el mundo de la investigación. Por guiarme en los estudios y aconsejarme. Gracias a ti me vine a vivir a Madrid y pude continuar con mi formación.

En el apartado personal, quiero agradecer a mis padres todo lo que han sacrificado, ayudado y aconsejado para que pueda llegar hasta aquí. Estoy segura que mi padre va a disfrutar tanto como el que más allí donde esté viendo que termino esta etapa. A ellos les debo todo lo que soy y he conseguido. A mi hermano Mikel por todos los momentos que compartimos, por ser mi compañero de aventuras desde peque. A mis tíos y primos que siempre me han apoyado y nos han acompañado en los momentos más difíciles, en especial a Maika y Alberto. A mi pareja Fernando, por todos los consejos, el apoyo incondicional y por celebrar todos los buenos momentos conmigo, has sido un gran apoyo. A Fernando y Azucena por estar siempre ahí, escuchándome y apoyándome.

Resumen

Con el auge de Internet la cantidad de datos generados ha crecido exponencialmente, la necesidad de analizar esta información y la aparición de nuevas aplicaciones que explotan estos datos dio lugar al desarrollo de distintos sistemas para la gestión de grandes cantidades de datos. El modelo tradicional de gestión de datos (bases de datos relacionales) no resultaba adecuado para todas las aplicaciones, además de no escalar a grandes cantidades de datos. Por esta razón se diseñaron distintos tipos de gestores de datos siendo las bases de datos NoSQL el paradigma más conocido, aunque no el único. Dentro de este tipo de bases de datos se encuentran las bases de datos para grafos, orientadas a documentos, almacenes clave-valor y las orientadas a columnas. Otro modelo completamente diferente son los sistemas de streaming de datos; los cuales procesan los datos al vuelo, sin almacenarlos. Ambos tipos de gestores de datos ejecutan en sistemas distribuidos con el fin de procesar grandes cantidades de datos.

El objetivo de la tesis es el diseño de protocolos para la mejora del rendimiento de dos tipos de sistemas para la gestión de grandes cantidades de datos: una base de datos NoSQL (HBase) y un sistema de streaming de datos (UPM-CEP). Concretamente, en el caso de HBase se proponen protocolos para la carga de datos y equilibrado de carga. La optimización del proceso de carga de datos es fundamental para mejorar la disponibilidad del sistema (tiempo que el sistema está operativo). La carga de datos se paraleliza y tiene en cuenta la distribución de los mismos con el fin de ubicarlos en distintas máquinas y mejorar el rendimiento de las consultas. El equilibrado de carga distribuye los datos de manera uniforme entre los distintos nodos con el fin de paralelizar las consultas y mejorar el rendimiento (tiempo de respuesta y productividad). Los protocolos tienen en cuenta la arquitectura NUMA del sistema subyacente.

También se han propuesto protocolos para dotar de elasticidad a un sistema de streaming de datos. La elasticidad permite al sistema realizar scale-in, paralelizando consultas en una misma máquina, scale-out, distribuyendo consultas o paralelizándolas en distintas máquinas, sin parar el sistema.

En ambos casos los resultados se han validado en dos clústers con distinta arquitectura con una decena de nodos, así como en un superordenador, empleando benchmarks industriales, TPC-C e Intel HiBench.

Summary

With the rise of the Internet, the amount of data generated has grown exponentially, the need to analyse this information and the appearance of new applications that exploit this data led to the development of different systems for managing large amounts of data. The traditional data management model (relational databases) was not suitable for all applications, in addition to not scaling to large amounts of data. For this reason, different types of data management systems were designed, with NoSQL databases being the best known paradigm, although not the only one. Databases for graphs, document-oriented, key-value stores and column-oriented are NoSQL data stores. Another completely different model is the data streaming model; which process the data on the fly, without storing it. Both types of data management systems run on distributed systems in order to process large amounts of data.

The goal of this thesis is the design of protocols to improve the performance of two types of systems for managing large amounts of data: a NoSQL database (HBase) and a data streaming system (UPM-CEP). Specifically, in the case of HBase, protocols for data loading and load balancing are proposed. Optimizing the data loading process is essential to improve system availability (time that the system is operational). The data load is parallelized and takes into account their distribution in order to place them on different machines and improve query performance. Load balancing distributes data evenly across nodes in order to parallelize queries and improve performance (response time and throughput). The protocols take into account the NUMA architecture of the underlying system.

Protocols have also been proposed for providing elasticity to a data streaming system. Elasticity allows the system to scale-in, by parallelizing queries on the same machine, scale-out, by distributing queries or parallelizing them on different nodes, without stopping the system.

In both cases, the results have been validated in two clusters with different architecture with a dozen nodes, as well as in a supercomputer, using industrial benchmarks, TPC-C and Intel HiBench.

Índice general

Índice de figuras

ÍNDICE DE FIGURAS

ÍNDICE DE FIGURAS

Índice de tablas

Índice de Algoritmos

ÍNDICE DE ALGORITMOS

Capítulo 1

Introducción

Durante más de veinte años las bases de datos relaciones han permitido almacenar datos de manera persistente, sin evolucionar en cuanto a su sistema de almacenamiento y gestión de los datos. En los años 2000 las bases de datos relacionales seguían siendo muy utilizadas, pero con la aparición de Internet para uso comercial durante esta década, produjo que las bases de datos relacionales no fuesen capaces de almacenar la gran cantidad de datos que se comenzaban a generar. Con el tiempo el uso de Internet fue incrementado, aumentando la cantidad de usuario conectados y generando contenido. Para soportar el aumento del volumen de datos se encuentran dos soluciones, *scale up* o *scale out*. La primera de ellas, scale up, consiste en adquirir una máquina con mayor almacenamiento y procesamiento que permita manejar las grandes cantidades de datos generadas. El principal problema de esta solución es el gran coste que tiene adquirir máquinas con esas características a demás del límite de almacenamiento que supone, ya que el espacio de almacenamiento tanto en disco como en memoria principal y los procesadores son limitados. La segunda solución, *scale out*, consiste en un conjunto de máquinas de hardware sencillo con un coste no muy elevado que permite aumentar la capacidad de almacenamiento y procesamiento mediante la incorporación de nuevas máquinas al conjunto, clúster de máquinas.

Pero las bases de datos relaciones no están diseñadas para ejecutarse de manera eficiente en un clúster de máquinas. Por ello Google,desarrolló un sistema de almacenamiento de datos distribuido (Google File System [1]) cuya arquitectura y funcio-

namiento fueron publicados en 2003. En 2006 Apache Software Foundation,presentó Hadoop Distributed File System (HDFS). Igualmente, un sistema de almacenamiento distribuido inspirado en GFS bajo una licencia Apache de código abierto. Apache permitió que muchas empresas pudiesen adquirir la capacidad de almacenamiento necesaria mediante este software y un clúster de maquinas con hardware estándar.

Por otro lado, los datos que estaban siendo almacenados en estos sistemas debían de ser procesados de manera más eficiente ya que el acceso a los datos y su procesamiento era muy costoso sin un software específico. Para ello en 2004, Google presentó MapReduce [2], una herramienta basada en el paradigma de computación del mismo nombre, cuya estrategia es "divide y vencerás" y que permite realizar un preprocesamiento de los datos en los mismo nodos para luego agruparlos y generar la salida deseada. Tras ellos,Apache incorporó en 2008 MapReduce a su proyecto Hadoop, de nuevo distribuido bajo la misma licencia Apache de código libre.

De forma paralela, en 2004 Google desarrolló BigTable [4], una base de datos *NoSQL* de tipo clave-valor que utiliza como almacenamiento persistente GFS. Permitiendo de esta manera estructurar y realizar consultas sobre los datos almacenados en su sistema de almacenamiento distribuido. Unos años más tarde, en 2008, Apache presentó HBase utilizando como referencia BigTable para dar soporte a Hadoop. Pero no fue hasta 2009 cuando apareció un término el cual englobaría a todas aquellas bases de datos que no utilizan el modelo relacional y pueden ser ejecutadas en un clúster de máquinas asegurando un buen rendimiento, *NoSQL*. Estas bases de datos almacenan la información en pares de tipo clave-valor llamadas tuplas. Con la aparición de las bases de datos *NoSQL*, surgió un modelo de almacenamiento persistente políglota que permite la utilización tanto de las bases de datos relaciones como *NoSQL*.

De manera paralela surgieron una serie de herramientas como Aurora [6] en 2003 y su sucesor Borealis [7] en 2005 que permiten el procesamiento de datos en memoria antes de ser almacenados de manera persistente, Este tipo de herramientas son conocidas como *Stream Processing Engines*. En 2009 se desarrolló Spark [8] con el objetivo de ampliar Hadoop y reemplazar MapReduce utilizando la técnica de computación en

memoria mediante *micro-batching*, es decir, transformando colecciones de pares clave-valor. A pesar de que no consiguió reemplazar a MapReduce, sí que fue integrado por gran cantidad de proveedores de *Big Data* como Cloudera [9], Horton [10], SAP [11] y MapR [12].

Un año más tarde, en 2010, apareció Flink [13] el cual permite realizar transformaciones de manera iterativa a colecciones de datos mediante flujos de datos cíclicos. Las transformaciones se realizan mediante la ejecución de consultas que se encuentran en continua ejecución y las cuales son representadas como un grafo acíclico. En el Laboratorio de Sistemas Distribuidos de la universidad politécnica de Madrid desde 2010 se ha estado desarrollando UPM-CEP un sistema de procesamiento de datos en streaming el cual ha sido utilizado en el desarrollo de parte de las contribuciones de esta tesis.

1.1. Sobre esta Tesis

El principal objetivo de esta tesis es el desarrollo de distintas herramientas que permiten mejorar el rendimiento de Tecnologías destinadas a la gestión, procesamiento y análisis de datos. Se ha desarrollado por completo en el Laboratorio de Sistemas Distribuidos de la Universidad Politécnica de Madrid y ha sido incluida en los proyectos de investigación: CUMULONIMBO: High Scalable Transactional Multi-Tier Platform as a Service (FP7-257993), COHERENTPAAS: A Coherent and Rich PaaS with a Common Programming Model (FP7-611068), CLOUD4BIGDATA: Efficient Cloud and BigData Infraestructure(S2013/ice-2894), CLOUDDBAPPLIANCE: European Cloud In-Memory Database Appliance with Predictable Performance for Critical Applications (H2020-732051), UEDOS: INSIGHTS + INTERACTIVE PLATFORM AND COMMUNITY ENGAGEMENT TOOLS FOR URBAN ECONOMIC DEVELOPMENT ORGANIZATIONS (19153-A19) y BigDataStack (H2020-779747).

1.1.1. Contribuciones de la Tesis

Las principales contribuciones de la parte de Tecnologías para la Gestión y Procesamiento de Datos se ha realizado sobre HBase y son:

1. **Carga de datos de forma masiva en HBase.** Una herramienta basada en ImportTSV [14] que permite cargar datos desde ficheros de tamaños de Gigabytes incluso Terabytes a HBase generando la clave de cada una de las tuplas mientras se procesan las líneas del fichero a cargar. La descripción de las claves puede ser indicada por el usuario de tal manera que se ajusten a las consultas que van a ser realizadas sobre los datos posteriormente lo cual permite obtener un mejor rendimiento durante la ejecución de las mismas.

2. **Distribución de los recursos de máquinas con arquitectura NUMA.** Cada vez es más común encontrarnos máquinas con arquitecturas NUMA lo que permite distribuir los recursos de la máquina (cores y memoria) y asignarlos a los distintos procesos. Conociendo la funcionalidad que tiene cada una de los procesos a ejecutar en la máquina se puede distribuir los recursos de tal manera que cada uno tenga una cantidad de cores y memoria diferentes permitiendo aprovechar al máximo el potencial de la máquina. Para demostrar su funcionalidad se han distribuidos los recursos de todos los servicios requeridos durante el proceso de carga masiva de datos.

3. **Distribución de los recursos de YARN [15].** YARN es un software del *stack* de Apache Hadoop el cual permite distribuir los recursos asignados a cada uno de los procesos MapReduce. YARN es utilizado para el proceso de carga masiva de datos, se ha diseñado e implementado una herramienta que permite asignar la configuración adecuada a cada proceso map o reduce de tal manera que se aprovechen al máximo los recursos de *MapReduce* mejorando el rendimiento del proceso de carga masiva en más de un 10 % con respecto a la configuración predefinida de YARN.

4. **Equilibrado de datos de manera uniforme en HBase.** A pesar de que HBase tiene distintas herramientas para el balanceo de carga, memoria y consumo de CPU. Se ha implementado una herramienta que permite distribuir lo datos de manera uniforme entre todos los nodos del sistema para cada una de las regiones de las tablas almacenadas en HBase de manera independiente almacenen la misma cantidad de datos. Este modelo de distribución de los datos permite mejorar el rendimiento de la base de datos cuando se realizan consultas con acceso uniforme a los datos. Además permite al sistema configurarse para adaptarse al aumento de carga añadiendo nuevos nodos o eliminando nodos en caso de que sea necesario asegurando que tras el equilibrado de carga cada uno de los nodos va a tener la misma cantidad de información.

Las principales contribuciones de la parte de Tecnologías para análisis de Datos se han realizado sobre UPM-CEP [16] [17]. UPM-CEP tiene varios servicios: orchestrator, metricServer e instanceManager. Este último servicio es el encargado de ejecutar instancias de las sub-consultas las cuales son las encargadas de procesar las tuplas que recibe el sistema. Todas la aportaciones presentadas en esta tesis han sido realizadas para mejorar procesamiento de tuplas en el conjunto del sistema y para ello todas ellas se han aplicado en el servicio instanceManager. En particular las contribuciones permiten dotar al UPM-CEP de elasticidad dinámica.

1. **Sistema de migración de sub-consultas en UPM-CEP.** Se ha implementado una herramienta que permite mover las sub-consultas de unos instanceManager a otros. Cuando el sistema comienza a recibir una carga superior a la que está procesando si hay varias sub-consultas en un mismo instanceManager o el instanceManager en el que se está ejecutando la sub-consulta no tiene suficientes recursos y hay otros instanceManagers en el sistema con más recursos, UPM-CEP es capaz de mover la sub-consulta de un instanceManager a otro para asegurar que la carga que está recibiendo sea procesada sin detener el procesamiento de tuplas en el resto de instanceManagers.

2. **Sistema de *Scale Up* de sub-consultas en UPM-CEP.** Cuando el sistema recibe una carga superior a la que puede procesar con las distribución actual de las sub-consultas y cada uno de los instanceManagers están ejecutando un instancia de sub-consulta. Se ha implementado un proceso que permite a UPM-CEP detectar esta situación y distribuir la carga de la instancia que está saturada creando una nueva instancia en otro instanceManager del sistema que se encuentre disponible y sin ejecutar ninguna otra sub-consulta. Este proceso se realiza de manera dinámica y el resto de instancias de la sub-consulta pueden seguir procesando la carga de tal manera que no se bloquea el procesamiento de tuplas.

3. **Sistema de *Scale Down* de sub-consultas en UPM-CEP.** Como situación opuesta a la anterior, cuando el sistema comienza a recibir una carga inferior a la que está acostumbrado a procesar, los instanceManagers dejan de utilizar todos sus recursos ya que no los necesitan. Con el objetivo de reconfigurar la distribución de las sub-consultas se ha implementado un proceso que tras detectar esta situación, elimina una o varias instancias de las sub-consultas y distribuye la carga que estaba siendo procesada por las instancias eliminadas entre las que van a seguir en ejecución. Durante este proceso las instancias que no son eliminadas siguen procesando la carga que le corresponde y además son configuradas para comenzar a procesar la carga de las instancias eliminadas.

1.2. Organización del documento

El documento está organizado en los siguientes capítulos. El Capítulo 2 se introducen tecnologías para la gestión y procesamiento de datos que se encuentran en la actualidad, como han ido evolucionando las bases de datos para poder almacenar y manejar las grandes cantidades de datos que se generan día a día, hasta presentar HBase y todas las tecnologías software que permiten el correcto funcionamiento de esta base de datos NoSQL. También se presenta UPM-CEP, un sistema de procesamiento de datos en streaming, diseñado y desarrollado en el Laboratorio de Sistemas Distribuidos de

la Universidad Politécnica de Madrid, finalizando con las presentación de las máquinas con arquitectura NUMA. En el Capítulo 3 se presentan tres de las contribuciones realizadas sobre los sistemas de gestión y almacenamiento de datos. En concreto se presentan un conjunto de herramientas que permiten realizar la carga masiva de datos en HBase aprovechando al máximo los recursos que ofrecen cada una de las máquinas del clúster, además se muestran distintas herramientas ya existentes y se comparan en la evaluación de rendimiento para finalmente mostrar la mejora en el rendimiento del proceso de carga de datos obtenida con las contribuciones. El Capítulo 4 está dedicado al equilibrado de datos uniforme, una herramienta que permite distribuir los datos entre todas las máquinas del clúster de manera uniforme. Tras finalizar el equilibrado, el rendimiento de las consultas con acceso uniforme a los datos almacenados en HBase mejora de manera considerable. El capítulo 5 muestra las principales contribuciones realizadas en el desarrollo del UPM-CEP que permiten la reconfiguración de las consultas de forma dinámica para ajustarse a la carga que está recibiendo el sistema, junto con la evaluación de rendimiento de las mismas. Finalmente, el capítulo 6 resume las contribuciones y resultados obtenidos en esta tesis y se presentan las futuras líneas de trabajo.

1. INTRODUCCIÓN

Capítulo 2

Tecnologías para la Gestión y Procesamiento de Datos

A finales de los años 90 con la aparición de Internet y durante los años 2000 con el desarrollo de las páginas web se comenzaron a generar grandes cantidades de datos: links, redes sociales, actividades en logs, mapeo de datos. La cantidad de datos generada iba en aumento a la par que la cantidad de usuarios también crecía.

La distinta procedencia y formato de los datos generados junto con el aumento de la cantidad de los mismos hacía inviable que las máquinas utilizadas para las bases de datos relaciones pudiesen soportar la carga. Una de las soluciones que se planteó fue la utilización de máquinas más potentes, con más recursos pero estas máquinas son muy costosas, además de los límites de procesamiento, espacio en disco y memoria que tienen las máquinas actuales. La alternativa era utilizar varias máquinas más simples que trabajasen de manera conjunta solucionando así el problema del espacio y de los fallos de hardware, ya que si una máquina falla el sistema sigue funcionando con el resto.

Esta solución hizo aparecer otro problema y es que las bases de datos relaciones no son capaces de ejecutarse en un clúster de máquinas. De ahí que Google y Amazon desarrollasen BigTable [4] y Dynamo [18] respectivamente para dar solución a sus propios problemas de gestión de datos. Con el tiempo otras organizaciones se vieron en el mismo problema y es en ese momento cuando comenzaron a aparecer otras bases de datos siguiendo los cánones de Google y Amazon. Algunas de ellas

son: Voldemort [19], Cassandra [20], Dynomite [21], HBase [22], Hypertable [23], CouchDB [24], y MongoDB [25]. Este conjunto de bases de datos se les llamó bases de datos NoSQL. BigTable es una base de datos NoSQL de tipo clave-valor que utiliza como almacenamiento persistente Google File System (GFS) [1], un sistema de almacenamiento distribuido y replicado. Permitiendo de esta manera estructurar y realizar consultas sobre los datos almacenados en su sistema de almacenamiento distribuido. Apache en 2008 presentó HBase utilizando como referencia BigTable.

Por otro lado dentro del análisis de datos hay varios tipos: *batch*, *stream*, *micro-batching* y *predictive*. El análisis de datos por *batch*, permite el procesamiento de grandes cantidades de datos almacenados, un ejemplo es MapReduce [2]. El análisis de datos en *stream*, procesa los datos en memoria tan pronto como son recibidos o generado. Permite analizar flujos (streams) infinitos de datos, detectando anomalías o generando alarmas sin necesidad de almacenar los datos previamente. Algunas de las tecnologías para el *streaming* de datos son Apache Flink [13], Apache Storm [26], Apache Samza [27] y WSO_2 [28]. El tercer tipo, *micro-batching*, empaqueta los datos en pequeños bloques en memoria para después procesarlos sin ser previamente almacenados. Un ejemplo es Apache Spark [8], que agrupa los datos recibidos en un RDD (Resilient Distributed Dataset), una estructura de datos en memoria o en disco que permite almacenar datos, para luego ser procesados. Finalmente, el análisis predictivo utiliza conceptos estadísticos, de inteligencia artificial, *machine learning* y *data modeling* para predecir comportamientos futuros en base a datos históricos.

Los sistemas de procesamiento en *Stream* permiten monitorizar y analizar flujos continuos de datos generados por uno o varios sensores, tráfico de red o analíticas de bases de datos en tiempo real generando alertas u otros eventos más complejos, mediante el despliegue de consultas continuas. Puede ser utilizado en distintas áreas como monitorización de la actividad empresarial, gestión de proyectos de negocio, sistemas de control de tráfico y control de edificios por medio de vídeo.

Con el aumento de dispositivos de monitorización como son pulsómetros, sensores de movimiento, GPSs y cámaras de videovigilancia entre otros, ha surgido el concepto

de *Wide-Area Stream Processing*. Este concepto hace referencia al procesamiento continuo de eventos generados por diversos dispositivos que se encuentran geográficamente distribuidos y que más tarde son procesados en centros de datos. Con la aparición de *Internet of Things (IoT)* y su aplicación en diferentes sectores como el transporte, salud, ocio y domótica se ha incrementado el volumen de datos generados por pequeños dispositivos. Algunos motores de *Stream processing* que pueden ser desplegados en área extendida son Aurora [6] y su predecesor Borealis [7] y StreamCloud [17].

En los últimos años los sistemas con arquitecturas multiprocesador han ido mejorando para ofrecer un mejor rendimiento durante la ejecución de distintas tareas de manera simultánea. Estos tipos de arquitecturas se caracterizan por tener varios procesadores que comparten otros periféricos como es la memoria y los buses. La principal ventaja de estas máquinas es la posibilidad de asociar un proceso a un conjunto de procesadores y por ello a una sección de memoria. Esta ventaja se puede utilizar para obtener un mayor rendimiento en las bases de datos NoSQL y en los sistemas de procesamiento en streaming distribuidos.

A lo largo de este capítulo se va a presentar en detalle Google Bigtable, su arquitectura y funcionamiento y la versión en código abierto de Apache Hadoop, HBase. Además se va a presentar CEP-UPM un sistema paralelo y distribuido que permite el procesamiento flujos continuos de datos implementado en el Laboratorio de Sistemas Distribuidos de la Universidad Politécnica de Madrid. Finalmente se presentarán las arquitecturas multiprocesador, más en detalle las arquitecturas Non-Uniform Memory Access (NUMA) y cómo pueden ser utilizadas para asignar procesos a las distintas secciones de memoria.

2.1. Google File System

Google File System [1] es un sistema de ficheros distribuido implementado por Google al final de la década de los 90 para dar soporte de almacenamiento a todas las tecnologías que estaban surgiendo, las cuales necesitaban almacenar grandes cantidades de información. Este software fue diseñado con el objetivo de tener un sistema de

almacenamiento de datos tolerante a fallos que proporcionase una latencia baja en los procesos de lectura y escritura y que además fuese altamente disponible. GFS proporciona una interfaz similar a la de un sistema de ficheros centralizado pero internamente se despliega en un conjunto de máquinas las cuales pueden estar geo-distribuidas. Los datos almacenados en GFS son almacenados en ficheros estándard los cuales son divididos internamente en bloques que son replicados y cuyas réplicas almacenadas en distintos nodos del clúster para tolerar fallos.

De esta manera, GFS proporciona: 1) Tolerancia a fallos: La replicación de los ficheros permite que los datos siempre estén disponibles si falla una réplica o más del clúster. 2) Sensación de almacenamiento ilimitado: Se pueden añadir nuevos nodos al clúster aumentando de esta manera la capacidad de almacenamiento. 3) Almacenamiento de ficheros de gran tamaño: Permite almacenar ficheros de cientos de Gigabytes incluso de Terabytes al fragmentar los fichero en bloques los cuales son almacenados en varios nodos del clúster. 4) Interfaz similar a un sistema de ficheros centralizado: Toda la arquitectura y funcionalidad de GFS permanece oculta al usuario tras un cliente que proporciona la misma interfaz que un sistemas de ficheros como puede ser el de Linux.

2.1.1. Almacenamiento de datos en Google File System

Los datos son almacenados en ficheros de texto plano de Linux llamados *GFS Files*, internamente estos ficheros de texto son divididos en bloques (*chunks*) de tamaño fijo, 64 MB. Los bloques son almacenados en el sistema de ficheros de Linux y son replicados en distintos nodos del clúster. El nivel de replicación estándar es tres, aunque puede ser modificado, al igual que el tamaño del bloque que puede ser seleccionado por el usuario al cargar un fichero en GFS.

Réplicas

Las réplicas, con nivel de replicación tres, se distribuyen dentro del clúster de la siguiente manera: 1) la réplica principal se almacena en el nodos del clúster más cer-

cano al cliente que ha cargado el fichero. 2) La segunda réplica se almacena en otro nodo del cluster que se encuentre en el mismo rack o red que la réplica primaria. 3) La tercera réplica en caso de que el clúster esté desplegado en varios racks, se almacena en un rack diferente del que tiene las réplicas principal y secundaria. En caso tener un nivel de replicación superior el resto de réplicas se irán colocando aleatoriamente en el resto de nodos del clúster. De esta manera se asegura alta disponibilidad ya que en caso de fallo del nodo o del rack siempre va a haber una réplica disponible, si no se produce un fallo general del clúster.

Por otro lado si los nodos del clúster tienen más de un disco de almacenamiento y si el número de réplicas es superior al número de nodos, puede haber más de una réplica en el mismo nodo y estas se almacenarían en los distintos discos del nodo. De esta manera, si uno de los discos del nodo falla, el otro disco tendrá una réplica de los datos. En cambio si el número de réplicas es superior al número de nodos y discos, no sería de utilidad un nivel de replicación tan alta ya que habría más de una réplica por nodo y disco y en caso de fallo se perdería el acceso a esas replicas.

GFS mantiene siempre el nivel de replicación de todos los bloques, en caso caso de que una de las réplicas quede inaccesible por un fallo de red o porque el nodo que la servía ha fallado, automáticamente comienza a crear una réplica de los bloques que estaba sirviendo el nodo fallido en otros nodos disponibles en el clúster mediante la copia de una de las otras réplicas de los bloques.

Tamaño de bloque

El tamaño de bloque utilizado tiene un gran impacto en el rendimiento del sistema. Los usuarios acceden a los ficheros almacenados en GFS mediante un cliente. Cuando el usuario solicita acceder un fichero, el cliente solicita a GFS la localización de las réplicas de los bloques del fichero. Información que el cliente almacena para evitar futuras consultas y que solo volverá a consultar en caso de que cambie dicha información. Con ella el cliente accede directamente a los nodos que poseen los bloques del fichero que el usuario ha solicitado leer. El cliente va solicitando información a medi-

da que el usuario va leyendo el fichero, ya que necesita conocer la localización de las réplicas de los bloques siguientes.

Si el tamaño de bloque es muy pequeño en comparación con el tamaño del fichero almacenado, el número de bloques generados va a ser alto, y por tanto la interacción entre el cliente y GFS para conocer la localización de las réplicas de los bloques va a realizarse con más frecuencia si lo comparamos con un tamaño de bloque mayor que generará un menor número de bloques. Por ejemplo un fichero de 10 GB utilizando un tamaño de bloque de 64 MB va a generar 160 bloques por lo que el cliente solicitará en 160 ocasiones la localización de las réplicas de los bloques en caso de que el usuario acceda a todo el fichero. Si se utiliza un tamaño de bloque mayor, por ejemplo 1024 MB se generan 10 bloques por lo que solicitará en 10 ocasiones la ubicación de las réplicas de los bloques, siempre y cuando desee leer el fichero por completo.

Si se utiliza un tamaño de bloque que genere pocos bloques, el número de conexiones que tiene que mantener el cliente con los nodos que sirven los bloques es menor con respecto a un tamaño bloque menor. Siguiendo con el ejemplo del fichero de 10 GB, para el tamaño de bloque de 1024 MB se van a establecer 10 conexiones entre el cliente y los nodos que sirven los bloques a diferencia de las 160 conexiones que se tienen que establecer con un tamaño de bloque de 64 MB.

Otro aspecto que se ve afectado por el tamaño de bloque es el acceso al mismo bloque por varios clientes de manera simultánea. Si el tamaño de bloque es grande, el número de clientes que accedan a un mismo bloque a la vez es mayor pudiendo llegar a colapsar el nodo que sirve el bloque al no ser este capaz de manejar un número tan alto de conexiones. En cambio si el tamaño de bloque fuese menor, el mismo contenido que se encontraba en el bloque inicial va a estar distribuido en varios bloques y en varios nodos del clúster permitiendo distribuir las conexiones de los clientes entre los distintos nodos.

2.1.2. Componentes de Google File System

La arquitectura de GFS está basada en tres servicios: uno o varios clientes o *GFS client*, un único *GFS Master* y uno o varios *GFS Chunkserver*. El *GFS client*, permite al usuario final interactuar con el sistema de manera transparente, como si estuviese trabajando con el sistema de ficheros de Linux, ya que permite realizar las operaciones de lectura, escritura y eliminación de ficheros. El servicio *GFS Chunkserver* almacena los bloques de los ficheros y ejecuta las operaciones necesarias sobre los mismos. Y el servicio *GFS Master* es el encargado de controlar el estado de los *GFS Chunkservers* y almacenar los metadatos de los ficheros almacenados en el sistema como la información referente a los bloques de los ficheros y la localización de las distintas réplicas.

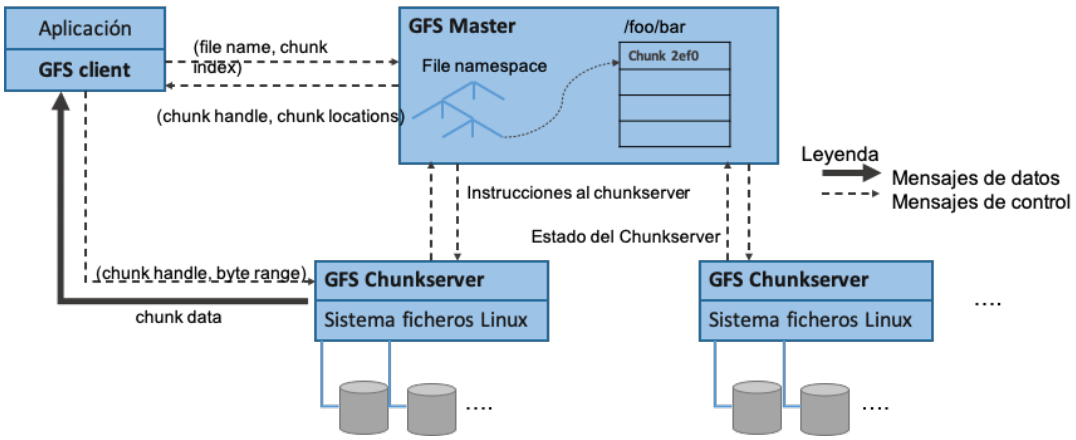


Figura 2.1: Arquitectura de GFS [1]

La figura 2.1 muestra la arquitectura de GFS, los distintos servicios, la interacción entre ellos y el flujo de las operaciones de lectura. En la parte superior izquierda de la figura 2.1, una aplicación está utilizando el cliente, *GFS client*, para interacturar con el sistema de ficheros. El *GFS client* se comunica con el *GFS Master* y los *GFS Chunkserver*, para ello utiliza dos tipos de mensajes diferentes, los mensajes de control y de datos. Los mensajes de control intercambiados entre *GFS client* y *GFS Master*, *(file name, chunk index)*, especificando el nombre del fichero y el indice del bloque a leer y *(chunk handle, chunk location)*, devolviendo el controlador del bloque a leer

y la localización del mismo, en la figura 2.1, permiten al *GFS client* conocer en qué *GFS Chunkserver* se encuentran las réplicas de los bloques que la aplicación quiere acceder. Esta información la almacena el *GFS client* para evitar futuras interacciones con el *GFS Master* para preguntar la misma información. Una vez que el *GFS client* conoce la localización de los bloques, mediante un mensaje de control *(chunk handle, byte range)* contacta con el o los *GFS Chunkservers* que sirven los bloques que poseen los datos a los que desea acceder la aplicación y estos mediante mensajes de datos envían los datos de los bloques solicitados al cliente.

Internamente, entre el *GFS Master* y los *GFS Chunkservers* se produce un intercambio de mensajes de control: 1) *Instructions to chunkservers*: El *GFS Master* indica a los *GFS Chunkservers* operaciones sobre los bloques que están sirviendo como puede ser, crear una nueva réplica de uno de los bloques, la migración de los bloques de un *GFS Chunkservers* a otro y información sobre el control de acceso a los bloques. 2) Periódicamente los *GFS Chunkservers* comunican su estado al *GFS Master*, especificando la información de los bloques que están sirviendo para que el *GFS Master* conozca en todo momento el estado del sistema.

2.1.3. Flujo de las operaciones escritura

El proceso de escritura require del intercambio de varios mensajes entre las distintas réplicas para asegurar que todas escriben los datos. En primero lugar y tras consultar el cliente la ubicación de las réplicas al *GFS Master*, este envía los datos a todas la réplicas. Un vez que el cliente ha terminado de enviar los datos, manda un mensaje a la réplica primaria para indicarle que puede comenzar con la escritura en disco de los datos. Una vez que la réplica primaria ha terminado de escribir los datos en disco, les asigna un número de serie y se lo envía al resto de réplicas. Este número de serie se utiliza para identificar de manera única la escritura y de este modo tener localizados los datos que ha sido almacenados en las réplicas en caso de fallo para poder deshacer los cambios que se hayan producido en las réplicas. Cuando las réplicas terminan de escribir los datos, asignan el número de serie y avisan a la réplica primaria de que

han terminado la escritura. Una vez que la réplica primaria ha recibido todos los mensajes de las réplicas, le avisa al cliente de que el proceso de escritura ha terminado correctamente.

La figura 2.2 muestra los pasos de la operación de escritura en un clúster con nivel de replicación 3: Réplica Primaria, Réplica Secundaria A y Réplica Secundaria B. Los pasos 1 y 2 representan la comunicación entre el GFS client y el GFS Master para obtener la ubicación de las réplicas. Seguido y mediante la utilización de mensajes de datos, el cliente envía los datos a cada una de las réplicas, paso 3. Cada una de las réplicas almacena los datos en un buffer LRU, una política de reemplazamiento de caché en la que se descartan los datos más antiguos para almacenar los más recientes. Un vez el cliente ha terminado de enviar los datos, este envía un mensaje a la réplica primaria para que inicie el proceso de escritura, paso 4.

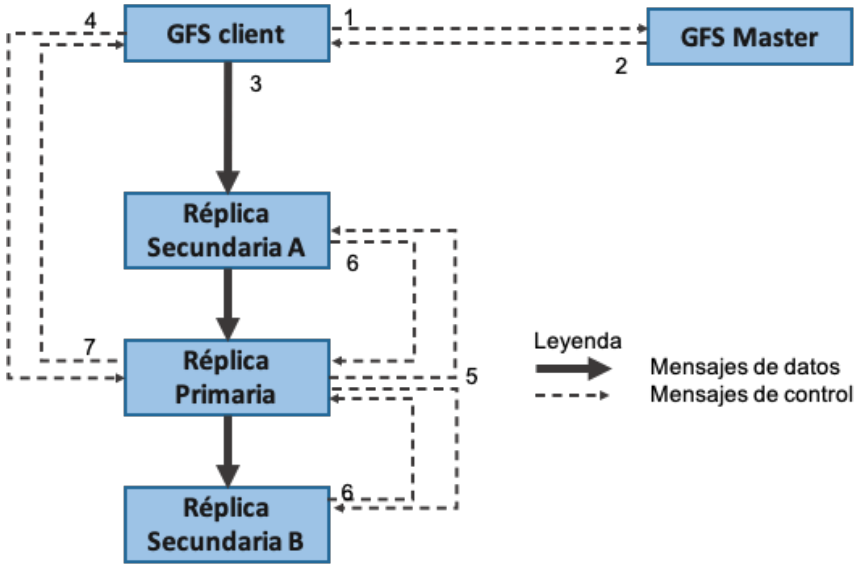


Figura 2.2: Flujo del proceso de escritura en GFS [1]

La réplica primaria asigna un número de serie a los datos que ha escrito en el bloque y lo envía al resto de réplicas, paso 5. A su vez cada una de las réplicas escriben los datos en el bloque, le asignan el número de serie recibido e informan a la réplica primaria de que han terminado, paso 6. Para finalizar, una vez que la réplica ha recibido

la confirmación del resto de réplicas, ésta comunica al cliente que ha finalizado la escritura, paso 7.

Durante el proceso de escritura pueden producirse fallos que impidan que el proceso de escritura termine correctamente. Dependiendo del momento en el que ocurra el fallo, GFS actúa de diferente forma. 1) El fallo se produce durante el proceso de escritura desde el buffer a disco en alguna de las réplicas secundarias. En este instante, la réplica primaria ya ha escrito los datos y les ha asignado un número de serie que es el que han recibido las réplicas secundarias para poder comenzar con la escritura. Al producirse el fallo en alguna de las réplicas secundarias, primero se informa al cliente del fallo y la réplica primaria vuelve a escribir de nuevo los datos asignándoles un nuevo número de serie. Este número es enviado a las réplicas las cuales van a proceder con la escritura de los datos y tras finalizar le asignarán el nuevo número de serie. Es decir se repiten los pasos 5 a 7 de la figura 2.2, hasta conseguir que todas la réplicas finalicen la escritura en disco. Las réplicas que habían conseguido escribir los datos con el número de serie anterior, invalidan los datos con dicho número de serie, ya que solo mantienen como válidos aquellos que se han escrito con el último número de serie válido. 2) El fallo se ha producido durante la escritura de los datos en la réplica primaria. En este caso, al igual que en el anterior se informa al cliente del fallo. Al no haber terminado el proceso de escritura en la réplica primaria, no se ha generado el número de serie y este no se había enviado a las réplicas secundarias. Por ello, se vuelven a escribir los datos en la réplica primaria la cual generará un número de serie tras finalizar la escritura. Una vez se ha escrito los datos en la réplica primaria de manera correcta, se enviará el número de serie al resto de réplicas para que continúen con el proceso de escritura. En la réplica en la que se ha producido el fallo, ya sea la réplica primaria como secundaria, los datos escritos la primera vez antes del fallo quedan invalidados por los escritos la siguiente vez ya que estos últimos son señalados con el número de serie y los anteriores no han llegado a tener asignado el número de serie.

2.2. MapReduce

El paradigma *MapReduce* [2] es un modelo de programación que permite el procesamiento de grandes cantidades de datos mediante la implementación de dos funciones. La función *map* transforma un conjunto de datos de entrada en formato clave/valor en otro conjunto de pares clave/valores donde valores es un conjunto de valores calculados durante la función map y cuya clave es igual. Estos pares ordenados por la clave constituyen la entrada de la otra función, la función *reduce*. La función *reduce* proceso los pares de entrada y aplica una serie de transformaciones que permiten generar un conjunto final de pares clave/valor. La principal ventaja de este paradigma es que permite ejecutar la función *map* de manera paralela sobre cada uno de los bloques de los ficheros almacenados en GFS para obtener unos datos intermedios que permitan un procesamiento más eficiente del total de los datos durante la función *reduce*. Este paradigma fue utilizado con anterioridad por otros lenguajes funcionales como Lisp pero en 2004 Google lo utilizó para implementar una librería la cual le permitía procesar los datos que estaban almacenados en GFS.

2.2.1. Funcionamiento de MapReduce

MapReduce procesa los ficheros almacenados en GFS. Por cada bloque de un fichero se ejecuta una tarea *map*. Esta tarea *map* procesa cada una de las líneas del bloque siendo cada una de ellos un par clave/valor donde la clave es el identificador de la línea y el valor es la línea a procesar. Por cada línea genera tantos pares clave/valor como lo requiera la función *map*. Los pares clave/valor de todas las tareas *map* son agrupados por clave, por ejemplo, si alguno de los pares de una tarea *map* tiene clave "A" y alguno de los pares de salida de otra tarea *map* tienen la misma clave, esto van a ser agrupados de tal manera que la clave seguirá siendo "A" y los valores serán la concatenación de los valores de los pares de ambas tareas *map*. Una vez se han agrupado los pares, comienza la ejecución de las tareas *reduce*. La tarea *reduce* va a generar un fichero de salida en el cual va a almacenar los pares clave/valor generados durante el procesamiento de los pares clave/valor de entrado por la función *reduce*.

La figura 2.3 muestra gráficamente el ejemplo *"WordCount"* presentado en [2], en el que se presentan las funciones map y reduce para contar el número de apariciones de cada una de las palabras de un texto.En este caso se utiliza una muestra de un texto (punto 1 de la figura), cada una de las líneas del fichero son pares clave/valor que son la entrada de la función *map* donde la clave es el número de línea y el valor es la línea. La función map, por cada línea va a obtener las palabras y va a generar tantos pares clave/valor como palabras, donde la clave es la palabra y el valor es un contador de la aparición, es decir 1 (puntos 2 y 3). Los pares son ordenados y agrupados por la clave, de tal manera que la clave es la palabra y el valor es un listado de 1 (punto 4). Estos nuevos pares ordenados son la entrada de la función *reduce*. Esta va a contar el número de unos de la lista, lo que va a permitir generar los pares clave/valor finales en los que la clave es la palabra y el valor es el número de apariciones (puntos 5 y 6).

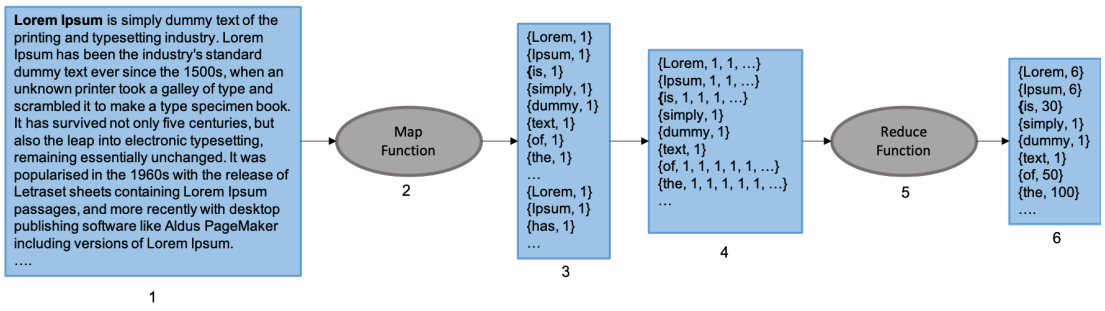


Figura 2.3: Ejemplo MapReduce

2.2.2. Procesos MapReduce

La ejecución de *MapReduce* da lugar a dos tipos de procesos que permiten la ejecución de las tareas *map* y *reduce*: el *MapReduce master* y los *MapReduce workers*. El proceso *MapReduce master* es un proceso único que controla la ejecución de las tareas *map* y *reduce*, es el encargado de asignar las tareas *map* y *reduce* a los diferentes *MapReduce workers*. Durante la ejecución de las diferentes tareas los procesos *MapReduce workers* envían su estado al *MapReduce master.* Los *MapReduce workers* son los procesos que se encargan de la ejecución de las tareas *map* o *reduce*.

La figura 2.4 muestra el flujo de una ejecución *MapReduce*: 1) El programa usuario instancia tantas tareas *map* cómo bloques del fichero. 2) El *MapReduce master* asigna cada tarea *map* a los *MapReduce workers* que se encuentren disponibles teniendo en cuenta la localización de las réplicas de los bloques a procesar. Es decir, las tareas *map* primero se ejecutarán en aquellos *MapReduce workers* que estén colocados en la misma máquina en la que esté almacena una réplica del bloque del fichero que se va a procesar. En caso de que no haya disponible ningún *MapReduce workers* colocado en la misma máquina, la tarea será asignada a uno que se encuentre en una máquina lo más cercana posible a una de las réplicas para reducir al máximo el tráfico de red producido al enviar el contenido del bloque a la tarea *map*. 3) El *MapReduce worker* ejecuta la función *map* implementada por el usuario sobre un bloque del fichero de entrada. Los pares clave/valor definidos por el usuario son almacenados en memoria. 4) Periódicamente esos pares almacenados en memoria son escritos en el disco local en R ficheros. La localización de esos ficheros es enviada al *MapReduce master*, el cual va enviar dicha información a las tareas *reduce*. 5) Cuando las tareas*reduce* conocen la localización de los ficheros intermedios, estos son procesados de tal manera que los pares obtenidos de las salidas de los procesos *map* son agrupados. Es decir, todos los pares con la misma clave se van a agrupar en un solo par en el cual se mantiene la clave y el valor es un conjunto de valores. Estos nuevos pares son ordenados lexicográficamente por la clave. 6) Las tareas *reduce* iteran sobre los nuevos pares ejecutando la función reduce por cada par y procesando el conjunto de valores del par. Cada reduce almacena los pares clave / valor generados en un fichero dando lugar a tantos ficheros cómo tareas *reduce* se ejecuten. El número de tareas reduce, se define en la configuración del trabajo realizada por el cliente antes de lanzar el trabajo *MapReduce*. 7) Cuando todas las tareas *map* y *reduce* han terminado el *MapReduce master* informa al cliente de que el proceso MapReduce ha terminado.

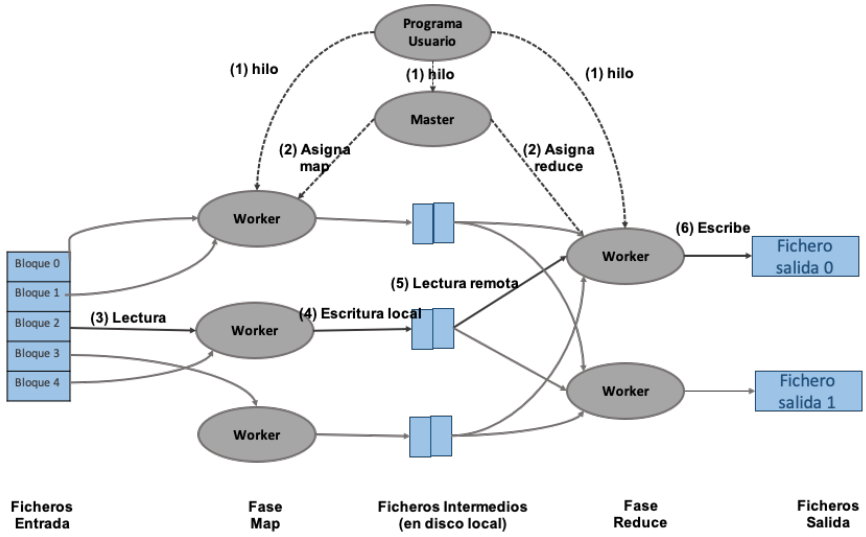


Figura 2.4: Esquema de la ejecución de un trabajo MapReduce [2]

2.2.3. Fallos durante la ejecución de un trabajo *MapReduce*

Para controlar los distintos tipos de fallos que pueden llegar a producirse durante la ejecución de un trabajo *MapReduce*, se utiliza el método de la replicación de tareas. Los posibles fallos que pueden llegar a producirse son: 1) fallo del nodo, el nodo por completo deja de funcionar. 2) Pérdida de comunicación entre el nodo y el resto del clúster, puede darse por un fallo en el switch o un particionado de red. 3) Fallo o falta de espacio en el disco que contiene la réplica del bloque a procesar o de los fichero intermedios. 4) El nodo tienen un rendimiento inferior al resto de nodos del clúster dificultando la finalización del trabajo.

Al estar los fichero almacenados en GFS esto se encuentran divididos en bloques los cuales a su vez se encuentran replicados y distribuidos en distintos nodos del clúster. Esta propiedad de los ficheros permite a *MapReduce* ejecutar de manera paralela varias tareas *map* o *reduce* que procesen el mismo bloque utilizando distintas réplicas.

En una primera instancia el *MapReduce master* lanza un única tarea *map* por bloque. En caso de que queden recursos disponibles en el clúster para lanzar más tareas *map*, el *MapReduce master* lanza nuevas tareas *map* que realicen el mismo trabajo que las iniciales utilizando otras réplicas del bloque. Lo mismo ocurre con las tareas *redu-*

ce, el *MapReduce master* lanza tantas tareas *reduce* como haya indicado el usuario en la configuración del trabajo y en caso de haber recursos suficientes se lanzarán nuevas tareas *reduce* que procesen los mismo datos que las tareas iniciales.

En caso de que las tareas terminen de manera correcta y no se produzca ningún fallo, la tarea que haya terminado en primer lugar avisa al *MapReduce master* y este se encarga de detener el resto de ejecuciones y de liberar los recursos para que puedan ser asignados a otras tareas. De esta manera, aunque ocurra alguno de los fallos comentados, siempre va a haber una tareas que termine correctamente siempre y cuando no se produzca un fallo general del cluster en cuyo caso el trabajo *MapReduce* no podrá terminar.

2.3. Chubby

Chubby [3] (2006) es un sistema de sincronización distribuido, altamente disponible y tolerante a fallos que es utilizado por varias aplicaciones de Google. Por ejemplo, GFS utiliza Chubby para elegir cual será la réplica primaria entre todas las disponible o para almacenar información que requiere alta disponibilidad, como por ejemplo los metadatos.

Internamente Chubby tiene una estructura jerárquica en forma de árbol que permite almacenar pequeñas cantidades de datos en memoria. El árbol está formado por una serie de directorios nodos, siendo en estos últimos donde se almacena la información.

Los nodos pueden ser efímeros o permanentes. El tipo se especifica durante la creación del nodo y se diferencian en la durabilidad del nodo en el sistema. Los nodos efímeros se eliminan tan pronto como el usuario que crea el nodo pierde conexión con el sistema. En cambio, si el nodo es permanente, permanece en el sistema hasta que sea borrado.

El acceso a la información se realiza mediante una librería que permite crear, leer, escribir y borrar datos y nodos entre otras acciones. Además, Chubby permite asignar cerrojos y suscribirse a eventos en los nodos y directorios. Por un lado, los cerrojos aseguran al cliente que lo a asignado que es el único que puede realizar cambios en el

nodo o en los niveles inferiores al directorio especificado. Y la suscripción a eventos notifica al cliente que lo ha solicitados cuando se han producido cambios en el nodo o directorio indicado. Los distintos eventos a los que un cliente puede suscribirse son: 1) modificaciones en el contenido de un nodo. 2) Creación, borrado o modificación de nodos en un directorio; 3) fallo del proceso *Chubby master*; 4) El cerrojo de un nodo o directorio queda invalidado, puede darse cuando hay problemas de conexión entre los nodos. 5) Un cliente ha adquirido un cerrojo en un nodo o directorio y 6) un cliente intenta adquirir un cerrojo en un nodo o directorio que ya lo tiene asignado a otro cliente.

2.3.1. Arquitectura de Chubby

Chubby tiene dos componentes que se comunican entre ellos mediante comunicación RPC (Remote Process Call o llamada a proceso remoto): la librería cliente y los servidores o réplicas. En la figura 2.5 se muestra la estructura de Chubby y la interacción entre ambos servicios. En la parte izquierda de la figura 2.5 aparecen las distintas aplicaciones, las cuales utilizan la librería cliente para comunicarse con Chubby y en la parte derecha se muestra un conjunto de servidores o réplicas llamado *"Cell"*, normalmente son 5 las réplicas que forman el *"Cell"*. Este número de réplicas hace quórum, es decir, permite obtener el número de votos suficiente para llevar a cabo una votación. Cuando todas las réplicas están activas, éstas comienzan el proceso de elección de la réplica máster. Para ello, cada una de las réplicas seleccionan como máster una de las réplicas del *"Cell"* y aquella que tenga una mayoría de votos será la elegida como máster.

Todas las réplicas mantienen una copia exacta de los datos pero únicamente la réplica máster es la encargada de iniciar las lecturas y escrituras de datos. Los clientes envían un mensaje a cualquiera de las réplicas disponibles en el *"Cell"* para preguntar por la réplica máster. En cuanto obtiene dicha información, el cliente realiza las peticiones a esta réplica. Las escrituras son propagadas a todas las réplicas desde la réplica máster y se informa al cliente que ha finalizado cuando la mayoría de las réplicas han

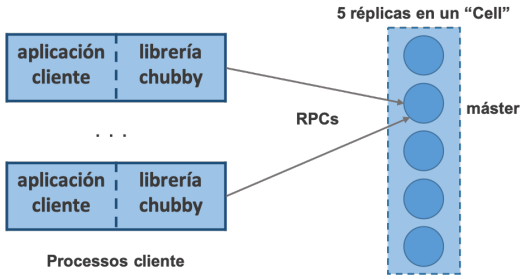


Figura 2.5: Estructura de Chubby [3]

terminado la escritura. Las lecturas son procesadas por el mismo máster. En caso de que la réplica máster falle, el resto de réplicas del *“Cell”* comienzan el proceso de elección de la réplica máster entre todas las que están disponibles.

2.4. Bigtable

Bigtable [4] es un sistema de almacenamiento distribuido para manejar estructuras de datos y que está diseñado para escalar en gran tamaño: petabytes de datos en miles de servidores con hardware sencillo. Otros productos de Google como Google Earth y Google Analitics utilizan Bigtable para almanacenar datos teniendo cada una de ellas distintos requisitos de espacios de almacenaje y tiempos de respuesta. Internamente Bigtable utiliza como sistema de almacenamiento persistente GFS y Chubby como almacenamiento de metadatos.

2.4.1. Organización de los datos

Bigtable es una matriz ordenada, dispersa, distribuida, persistente y multidimensional. La matriz es indexada por el triplete *clave de fila : clave de columna : marca de tiempo (timestamp)* y cada valor almacenado en la matriz es una cadena de bytes.

Las claves de la fila son un conjunto de caracteres que identifican de manera única cada una de las filas de la tabla. Las filas dentro de una misma tabla se encuentran ordenadas de manera lexicográfica teniendo en cuenta la clave de la fila.

Las claves de columnas son agrupadas en conjuntos llamados “familias de columnas”. Las “familias de columnas” tienen que ser definidas durante la creación de la tabla y no pueden ser modificadas después. Las claves de columnas se identifican utilizando la siguiente sintaxis: *familia de columnas : columna*. El nombre de la familia de columnas tienen que ser único en la tabla y el nombre de la columna tienen que ser único dentro de las columnas de la “familia de columnas”. Las columnas se crean al realizar un inserción de un dato en la tabla. Puede darse el caso que para una clave de fila no se inserte ningún dato en una clave de columna, en ese caso no ocupa espacio de almacenamiento.

Cada par *clave de fila : clave de columna* identifica de manera única una celda. Las celdas pueden guardar varias versiones del mismo dato, cada una de ellas es identificada por el timestamp. El timestamp es un número entero de 64 bits que puede ser la marca de tiempo cuando se ha realizado la inserción del dato o un número creciente asignado en el momento de la inserción.

La figura 2.6 muestracómo son almacenadas las versiones de los datos de una tabla. La cantidad de versiones a almacenar puede ser definida por el cliente indicando que desea almacenar las N últimas versiones o que desea almacenar las versiones más recientes desde un punto, por ejemplo las versiones de la última semana. Esta configuración se realiza a nivel de familia de columnas. Como se muestra en la figura 2.6, la familia de columnas FC-1 permite almacenar las 3 versiones más recientes de sus columnas, la familia de columnas FC-2 solo almacena la versión más actual del dato y la familia de columnas FC-n almacena las dos últimas versiones. Los valores de las celdas “Clave-n : “FC-1:C-2” y “Clave-n : “FC-2:C-4” se han insertado simultáneamente utilizando el mismo *timestamp* ts1, el resto de valores en las celdas “Clave-n : “FC-1:C-2””, “Clave-n : “FC-2:C-4””y “Clave-n : “FC-n:C-r”” han sido insertados en distintos momentos ya que los *timestamps* son diferentes. Cuando el cliente solicita un dato, si no especifica la versión durante la petición, devuelve el valor de la versión más reciente, por ejemplo, “x1” para la celda “Clave-n : “FC-1:C-2” : ts20”. Si se desea obtener todas las versiones de un dato o una versión en concreto, el cliente tiene que indicarlo.

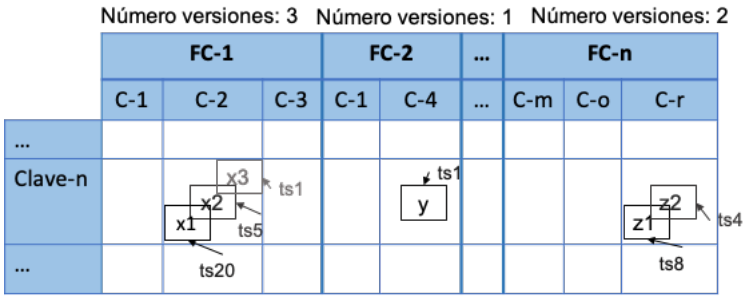


Figura 2.6: Manejo de las versiones en HBase

Cuando el tamaño de la tabla aumenta, ésta es particionada horizontalmente, esta partición de claves de filas ordenadas se llama *tablet*. Los *tablets* se usan en Bigtable para distribuir los datos y la carga entre distintos nodos.Los *tablets* se identifican por el par [clave inicio - clave fin) siendo la clave-inicio la primera clave de la fila del *tablet* y la clave-fin quedando excluida del *tablet*. En el caso del primer y último *tablet* de la tabla, la clave de inicio y la clave de fin respectivamente no son definidas. Esto es debido a que siempre es posible añadir una nueva fila con una clave menor o mayor que el primer o último *tablet* respectivamente.

La figura 2.7 muestra un ejemplo de una tabla que con n familias de columnas, en la que se han insertados filas con claves desde clave-a a clave-z y la tabla ha sido particionada en diferentes *tablets*.En la figura el primer *tablet* está delimitado por [- clave-j), el siguiente por [clave-j - clave-p) donde la fila con la clave clave-j pertenece al segundo *tablet* y no al primero de ellos. Los *tablets* primero y último de la tabla tienen delimitadores deferentes, la clave inicial del primer *tablet* y la clave final del último *tablet* están vacíos.

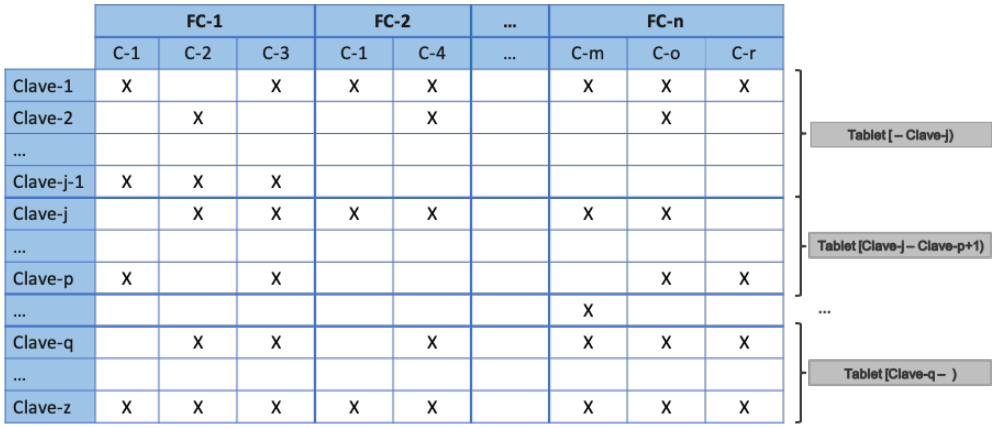


Figura 2.7: Organización de los datos en HBase

2.4.2. Arquitectura de Bigtable

Bigtable consta de tres componentes: la librería cliente, un *Bigtable master* y varios *Bigtable tablet servers*. Los clientes, a través de la API facilitada por la librería cliente, pueden crear, eliminar tablas y familias de columnas, acceder a la información de los metadatos, modificar el clúster, tablas y familias de columnas. Además, los clientes también pueden leer, eliminar datos de una fila o conjuntos de filas. Todas estas operaciones son atómicas. Un conjunto de operaciones sobre una única fila son ejecutadas de manera secuencial.

Los *Bigtable tablet servers* son los encargados de realizar las operaciones sobre los *tablets*, cada uno de ellos es capaz de servir varios *tablets* de la misma o de diferentes tablas. Al menos tiene que haber un *Bigtable tablet servers* en ejecución en el sistema para su correcto funcionamiento. Este servicio puede ser incorporado o eliminado del clúster dinámicamente para adaptar el sistema a la carga que esta recibiendo. Los *Bigtable tablet servers* manejan las peticiones de lectura y escritura recibidas de los clientes y ejecutan todas las operaciones solicitadas por el servicio *Bigtable master* como puede ser crear o eliminar una tabla, particionar un *tablet*, mover un *tablet* a otro *Bigtable tablet servers* e indicar el conjunto de *tablets* que está sirviendo.

El servicio *Bigtable master* es responsable de la ejecución de diversas tareas: 1) Mantiene el estado del clúster, periódicamente los *Bigtable tablet servers* envían men-

sajes “estoy vivo” al *Bigtable máster*. 2) Almacena la localización de los tablets en los diferentes *Bigtable tablet servers*. 3) Detecta si un *Bigtable tablet server* ha dejado de servir un tablet y re-asigna el tablet a otro *Bigtable tablet server* disponible. 4) Particiona los tablets que exceden el máximo tamaño de tablet o si el cliente ejecuta el proceso de división manualmente. 5) Controla la carga de los *Bigtable tablet servers* y ejecuta el algoritmo encargado de distribuir la carga moviendo los tablets desde los *Bigtable tablet servers* con más carga a los que tienen menos carga. 6) Maneja los cambios en el esquema de las tablas, crea y elimina tablas y maneja la creación de familias de columnas.

En la figura 2.8, se muestra un ejemplo de como una tabla es distribuida entre los distintos *Bigtable tablet servers* disponibles en el clúster, en este caso tres: Tablet Server 1, Tablet Server 2 y Tablet Server 3. Supongamos que hay una tabla con 150 filas donde la clave es un array de bytes que representa valores de 1 a 150. El usuario ha decidido crear *tablets* de tal manera que cada una de ellos contenga 25 filas y los ha distribuido en round-robin entre los tres *Bigtable tablet servers*. Para realizar estas operaciones el usuario ha indicado al *Bigtable master* mediante la librería cliente las claves por las que se van a crear los *tablets* y es el *Bigtable master* el que ha indicado al *Bigtable Tablet Server* que estaba sirviendo el *tablet* que realice las particiones por las claves que ha indicado el usuario. Una vez ha terminado la creación de los nuevos *tablets*, estos permanecen *Bigtable Tablet Server* que estaba sirviendo el *tablet* inicial. Para distribuir los *tablets* entre los distintos *Bigtable Tablet Server* el usuario ha solicitado al servico *Bigtable master* que mueva los tablets desde el *Bigtable Tablet Server* a otro de los *Bigtable Tablet Servers* disponibles siguiendo una distribución round-robin. Es decir, si el *tablet* inicial se encontraba en el Tablet Server 1, el usuario ha indicado que el *tablet* [26-51) se mueva al Tablet Server 2, el *tablet* [51-75) se mueva al Tablet Server 3, el *tablet* [101-126) se mueva al Tablet Server 2 y el *tablet* [126-) se mueva al Tablet Server 3. De esta manera, el primer *tablet* está delimitado por las claves [-26), contiene las filas 1 a 25 y es controlado por el Bigtable Tablet Server 1, el segundo *tablet* está delimitado por las claves [26-51), contiene las filas 26 a 50 y es servido por el Bigtable Tablet Server 2, y así sucesivamente.

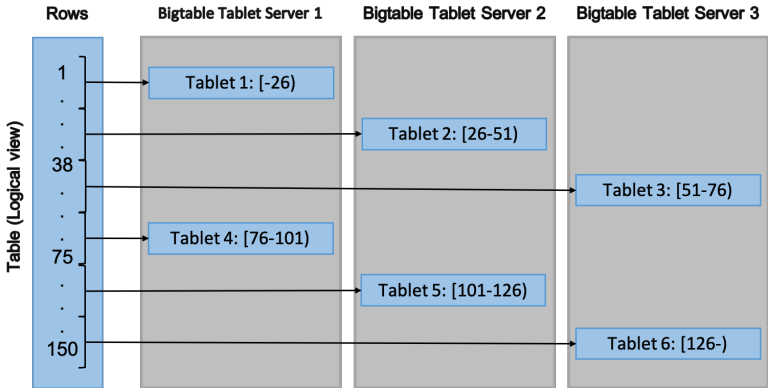


Figura 2.8: Distribución de una tabla entre los Bigtable tablet servers

Todas las operaciones de lectura o modificación de los datos almacenados en Bigtable requieren de un conocimiento previo por parte de la librería cliente para conocer qué *Bigtable Tablet Server* sirve el *tablet* que contiene los datos que desea acceder. Esta información la proporciona el *Bigtable master*, una vez el cliente recibe dicha información la almacena y no vuelve a consultarla al menos que al ir a realizar alguna operación esta falle porque la localización del *tablet* haya cambiado. En el resto de ocasiones, la librería cliente puede conectarse directamente con el *Bigtable Table Server* y realizar la operación que desee.

2.4.3. Persistencia de los datos en Google File System

Hasta ahora, se ha visto como se organizan los datos en Bigtable pero todos estos datos son almacenados de manera persistente para asegurar que los datos no se pierdan si el sistema se apaga o si ocurre algún tipo de fallo, para ello Bigtable utiliza GFS.

Los datos son almacenados en ficheros llamados *SSTable* en GFS. Por cada *tablet* y "familia de columnas" se crea un *SSTable*. Las operaciones de lectura y escritura requieren en algunos momentos del acceso a los *SSTables*. En un primer momento cada *Bigtable Tablet Server* almacena los datos en memoria en un buffer LRU llamado "memtable" y a su vez en un fichero en GFS llamado "tablet log". Por cada *tablet* que esté sirviendo el *Bigtable Tablet Server* hay un "memtable" y un "tablet log" asociado a él. El "tablet log" registra el historial de cambios para poder rehacer las operaciones

de escritura que no hayan sido almacenadas en los *SSTable* en caso de que el *Bigtable Tablet Server* falle. El "memtable" va almacenando las escrituras y en el momento en el que reemplaza los datos que más tiempo llevan en el "memtable" por los más recientes, los que han sido reemplazados son almacenados en los ficheros *SSTable* correspondientes. De esta manera si se tiene que recuperar un *tablet*, *Bigtable Tablet Server* tiene toda la información necesaria en el "tablet log" y en los ficheros SSTable.

Cuando se recibe una operación de lectura el *Bigtable Tablet Server* va a obtener los datos del "memtable" y en caso de que no estén o falten parte de los datos, leerá los datos de los *SSTables*.

La figura 2.9 muestra la representación de un tablet. En la parte superior muestra el buffer "memtable" que se encuentra en memoria y en la parte inferior el "tablet log" y los ficheros *SSTable* que se encuentran almacenados en GFS. Por otro lado muestra cómo las operaciones de escritura interactúan con el "tablet log" y el "memtable" para almacenan los datos simultáneamente y las operaciones de lectura acceden al "memtable" y los *SSTables* para obtener los datos.

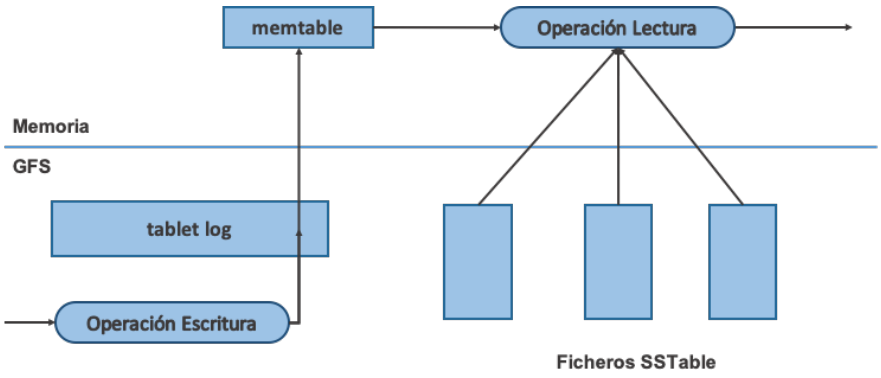


Figura 2.9: Representación de un *tablet* [4]

2.4.4. Interacción de Bigtable con Chubby

Bigtable utiliza Chubby para: 1) Asegurar que solo hay un proceso *Bigtable master* corriendo en el cluster; 2) El *Bigtable master* accede al directorio *servers* de Chubby para encontrar los *Bigtable tablet servers* que estén activos. 3) Para descubrir nuevos

Bigtable tablet servers y eliminar el registro de los que han fallado; 4) Almacenar los esquemas de cada una de las tablas de usuario; 5) Almacenar la lista de control de acceso. De ahí que el papel que desempeña Chubby en el correcto funcionamiento de Bigtable sea tan importante, que en caso de que falle durante un largo periodo de tiempo, Bigtable llega a dejar de estar disponible.

Localización de tablets Bigtable utiliza un árbol B+ con tres niveles para almacenar la localización de los tablets. El primer nivel se encuentra almacenado en Chubby, tal y como se muestra en la figura 2.10. En este fichero se almacena la localización de la tabla de metadatos principal llamada *Root Tablet*, esta tabla nunca es particionada en más tablets para mantener los tres niveles del árbol. La tabla *Root Tablet* almacena la localización de todos los tablets en una tabla especial de metadatos (*METADATOS de todos los tablets* en la figura 2.10). La tabla *METADATOS* almacena la localización de un tablet con una clave codificada de el identificador del *tablet* de la tabla y la clave de la fila final del *tablet*. Cada fila en la tabla *METADATOS* almacena aproximadamente 1KB de datos en memoria.

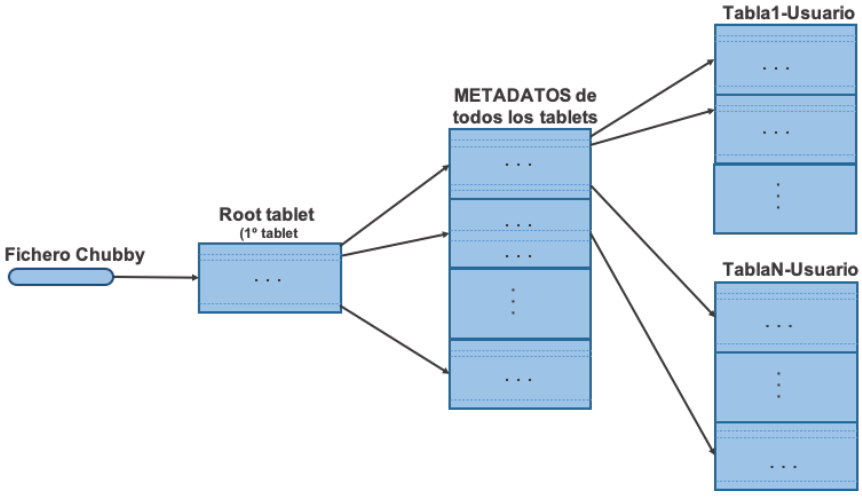


Figura 2.10: Jerarquía de la localización de tablets [4]

Control de *Bigtable tablet servers*

El servicio *Bigtable master* maneja la aparición de nuevos *Bigtable tablet servers* en el sistema a través de Chubby. Este servicio crea un directorio en Chubby llamado *servidores* en el que todos los *Bigtable tablet servers* van a crear un directorio persistente y van a asignar un cerrojo donde van a almacenar su información cuando arranquen. El *Bigtable master* está suscrito a cambios en este directorio, de esta manera detecta cuando un *Bigtable tablet server* se ha registrado en el sistema. Si el *Bigtable tablet server* pierde el cerrojo por un particionado de red u otro motivo, intenta conseguirlo de nuevo y en caso de no conseguirlo se para dejando de formar parte del clúster.

Las figuras 2.11 y 2.12 muestran gráficamente los pasos que se siguen al incorporarse un nuevo *Bigtable tablet server* y cuando falla uno perteneciente al clúster, respectivamente. En ambas figuras en la parte izquierda se muestra parte de árbol de Chubby y en la parte derecha los servicios involucrados, *Bigtable master* y *Bigtable tablet server*.

En un primer momento el *Bigtable master* crea el directorio *servidores* en Chubby y se subscribe a cambios para recibir una notificación si se crean nuevos nodos hijos, paso 1 de la figura 2.11. Cuando un nuevo *Bigtable tablet server* se incorpora al clúster crea un nodo permanente en el directorio *servidores* y coloca un cerrojo, paso 2. En ese momento el controlador del directorio *servidores* avisa al *Bigtable master* de que se ha creado un nuevo nodo hijo, paso 3 de la figura, y a partir de ese momento el *Bigtable master* comienza a intercambiar mensajes con el *Bigtable tablet server* para saber si el *Bigtable tablet server* mantiene el cerrojo en su nodo de Chubby, paso 4.

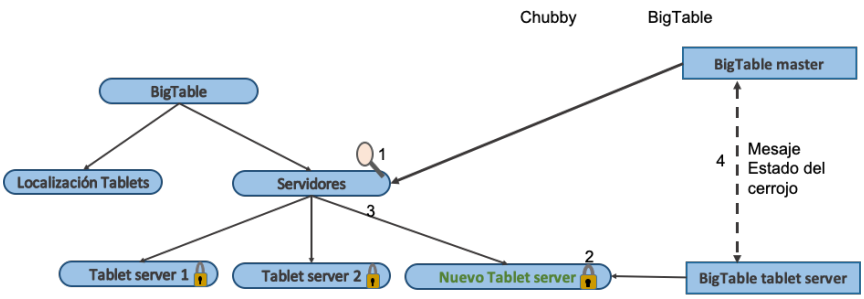


Figura 2.11: Descubrimiento de nuevos Bigtable tablet servers

En caso de que el *Bigtable tablet server* pierda el cerrojo, por ejemplo, si ha perdido conexión con Chubby y este sigue ejecutándose, va a intentar conseguirlo de nuevo, paso 1 figura 2.12. En el intercambio de mensajes entre el *Bigtable master* y el *Bigtable tablet server*, si el *Bigtable tablet server* sigue vivo le comunica al *Bigtable master* que ha perdido el cerrojo. A partir de ese momento el *Bigtable master* va a intentar obtener el cerrojo del nodo de ese *Bigtable tablet server*, pasos 2 y 3 de la figura. Si no puede obtenerlo, considera que Chubby no está disponible y tras un periodo de tiempo si no puede acceder a Chubby, Bigtable se volverá inaccesible. Pero si el *Bigtable master* puede obtener el cerrojo y sigue teniendo conexión con el *Bigtable tablet server*, considera que el *Bigtable tablet server* se encuentra en un estado inconsistente y le envía un mensaje al *Bigtable tablet server* para que se pare. A su vez elimina del directorio *servidores* el nodo que correspondía a dicho *Bigtable tablet server*, paso 4 de la figura 2.12. Si tras conseguir el cerrojo, no tiene conexión con el *Bigtable tablet server*, considera que ha fallado y directamente borra el directorio del *Bigtable tablet server*.

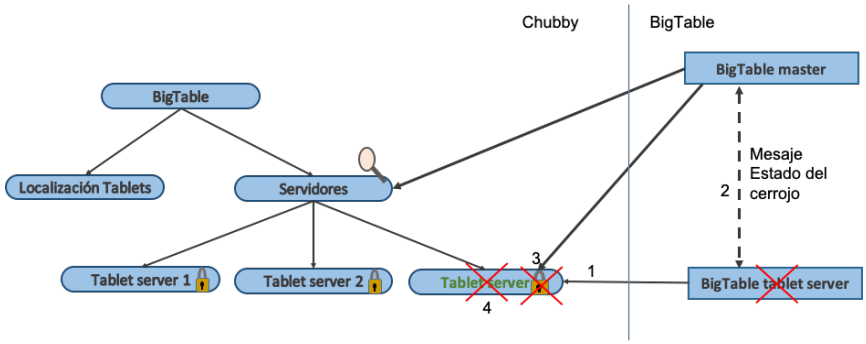


Figura 2.12: Fallo de un Bigtable tablet server

Cuando un *Bigtable tablet server* falla, el *Bigtable master* marca todos los tablet que estaba sirviendo dicho *Bigtable tablet server* como “no servidos” en la tabla de metadatos y a partir de ese momento comienza a distribuir los tablets entre los *Bigtable tablet server* disponibles.

2.5. Apache Hadoop

El software anteriormente presentado es propiedad de Google.Apache Software Fundation, inspirado por los artículos que publicó Google sobre sus diferentes herramientas, ha desarrollado un proyecto llamado Hadoop distribuido bajo la licencia de código abierto de Apache. Este proyecto está compuesto por diferentes herramientas entre las cuales se encuentran la versión en código abierto de las presentadas anteriormente. En la tabla 2.1 se muestra el software desarrollado por Google y su equivalente desarrollado por Apache Software Fundation.

Tabla 2.1: Hadoop Apache y su equivalente de Google

Google	Hadoop Apache
GFS	HDFS
MapReduce	MapReduce
-	YARN
Chubby	ZooKeeper
BigTable	HBase

2.5.1. Hadoop Distributed File System

Hadoop Distributed File System o HDFS [29] [30] es el sistema de ficheros distribuido de Apache, permite el almacenamiento de ficheros en un conjunto de máquinas de hardware estándar conectadas entre sí. Corresponde a la implementación de código abierto de GFS.

Los mismos componentes de GFS tienen otro nombre en HDFS, el GFS máster se llama HDFS Namenode y los GFS chunkservers son los HDFS DataNodes. Los GFS chunks son llamados en HDFS bloques y en este caso el tamaño de bloque predefinido es de 128 MB, valor que puede ser modificado por el usuario al igual que ocurre en GFS. El comportamiento de cada uno de los componentes, el manejo de los ficheros y el almacenamiento de los datos es exactamente igual que en GFS.

2.5.2. MapReduce y YARN

En 2011 se presentó la versión en código abierto de MapReduce. Esta implementación se basa en el artículo presentado por Google [31]. En este caso el proceso *MapReduce master* de la distribución de Google se llama *MapReduce JobTracker* y al igual que en la versión de Google se encarga de asignar las tareas de los trabajos a los *MapReduce workers*, además de controlar la ejecución de todos los trabajos. Los servicios *MapReduce workers* se llaman *MapReduce TaskManagers* y se encargan de llevar acabo las ejecuciones de las tareas *map* y *reduce*.

En 2013 Apache presentó YARN [15] [32] (*Yet Another Resource Negociator*), un gestor de recursos en clúster que permite gestionar de manera más precisa los recursos de CPU, memoria y disco. La principal deferencia con MapReduce es la división de las funcionalidades del servicio *MapReduce JobTracker* en dos servicios diferentes: *YARN ResourceManager* y *Application Master*. De esta manera la gestión de los recursos a nivel de general y la creación de los contenedores es llevada acabo por el servicio *YARN ResourceManager*, servicio que se ejecuta en una máquina a parte tal como sucede con el *MapReduce JobTracker*. Un contenedor es un conjunto de recursos (memoria, CPU, etc.) de un nodo particular donde se van a ejecutar las tareas *map* o *reduce*. El proceso *Application Master* se ejecuta en un contenedor especial creado por el *YARN ResourceManager* y es desplegado en uno de los *YARN NodeManagers* anteriormente llamados *MapReduce TaskTrackers*. El proceso *Application Master* es creado al inicio de la ejecución de cada uno de los trabajos MapReduce y controla de manera única todo el proceso de ejecución del trabajo para el que ha sido creado. Además, es el encargado de solicitar los contenedores necesarios para la ejecución de cada una de las tareas *map* o *reduce* al proceso *YARN ResourceManager*.

2.5.3. HBase

La versión de código abierto de BigTable, HBase, fue integrada dentro del proyecto Hadoop en 2008. HBase utiliza como almacenamiento persistente HDFS y como coordinador ZooKeeper [33], la implementación de código abierto de Chubby. Los

servicios de HBase son *HBase master* y *HBase region server* (HRS) y corresponden respectivamente con los servicios de Bigtable, *Bigtable master* y *Bigtable tablet server*.

2.5.3.1. Políticas de particionado

Al igual que Bigtable, HBase divide las tablas en regiones (tablets en Bigtable) mediante un conjunto de políticas de particionamiento que permiten dividir la tabla en regiones de manera automática. Cuando se crea una nueva región, ésta permanece en el mismo *HBase region server* en el que se encontraba la región original. Las políticas de particionamiento que dispone HBase son: 1) *IncresingToUpperBoundRegionSplitPolicy*: Es la política predefinida en HBase y entre todas las existentes es la más agresiva. Tiene en cuenta el número de regiones que se están sirviendo en el propio *HRS* y los valores asignados a las propiedades "hbase.hregion.memstore.flush.size" y "hbase.region.max.filesize" para calcular el tamaño máximo de las regiones antes de ser divididas. La propiedad "hbase.hregion.memstore.flush.size" indica el tamaño que tiene que tener el memstore (memtable en Bigtable) antes de copiar los datos en los ficheros HFiles (SSTable en Bigtable) y la propiedad "hbase.region.max.filesize" indica el máximo tamaño que puede tener una región. En cuanto la región alcanza el tamaño calculado, se divide la región en dos de tal manera que ambas tienen la misma cantidad de datos. Para ello se utiliza la siguiente fórmula: $Min(R^2 * "hbase.hregion.memstore.flush.size", "hbase.hregion.max.filesize")$, donde R es el número de regiones almacenadas en el mismo *HRS*. De esta manera, si tenemos en cuenta los valores predefinidos de la propiedades mencionadas (128 MB y 10 GB respectivamente), en un *HRS* que tiene una región. Esta región se dividirá tras alcanzar los 128 MB dando lugar a dos regiones de 64 MB. Ahora hay dos regiones por lo que cualquiera de ella se volverá a dividir en dos cuando alcance el tamaño de 512 MB, por lo que en ese momento habrá tres regiones. Según el número de regiones aumenta el tamaño que tiene que tener la región para dividirse será de 1152 MB con tres regiones, 2 GB con cuatro regiones y así sucesivamente, hasta alcanzar el tamaño equivalente al indicado en la propiedad "hbase.region.max.filesize". Continuando con este escenario, el tamaño es de 10 GB

y es alcanzado cuando el *HRS* está sirviendo 9 regiones. A partir de ese momento el tamaño máximo que las regiones van a alcanzar antes de ser divididas es de 10 GB.

2) *ConstantSizeRegionSplitPolicy*: Una región se divide en dos regiones del mismo tamaño cuando la región excede el máximo tamaño definido en la propiedad "hbase.hregion.max.
filesize". El punto de división al igual que con la política anterior se corresponde a fila que se encuentra en la mitad de la región original.

3) *DelimitedKeyPrefixRegionSplitPolicy*: Utiliza la primera parte de la clave hasta llegar a la primera aparición del prefijo como delimitador. Es decir, todas las filas cuya clave comience con el mismo patrón hasta la primera ocurrencia del prefijo indicado van a formar parte de la misma región. Por ejemplo, si las claves de las filas siguen el formato idUsuario_idAsignatura_idGrupo y el delimitador es '_', todas las filas con el mismo idUsuario van a pertenecer a la misma región. El principal inconveniente de esta política es que el usuario tiene que conocer a priori la composición de las claves que va a almacenar en la tabla.

4) *KeyPrefixRegionSplitPolicy*: Divide las regiones teniendo en cuenta el prefijo especificado por el usuario, todas las filas cuya clave comience con el mismo prefijo van a ser almacenadas en la misma región. Por ejemplo, si el usuario especifica los prefijos AA, BB, ..., todas las filas cuya clave comience por AA van a ir a una región, las que comiencen por BB irán en otra y así sucesivamente. Al igual que ocurre con la política anterior, el usuario necesita conocer previamente cómo son las claves, además es probable que las regiones queden desequilibradas.

2.5.3.2. Políticas de Distribución de la Carga

El rendimiento de HBase se puede ver seriamente afectado debido a una mala distribución de las regiones entre los *HRS*. Cuando un *HRS* tiene regiones que reciben peticiones con mucha frecuencia, este puede llegar a convertirse en un cuello de botella. HBase tiene predefinidos tres procedimientos de distribución de carga que permiten distribuir las regiones entre los *HRS* que no están soportando tanta carga, moviendo

las regiones desde los más cargados a los *HRS* con menos carga. Cada cinco minutos el distribuidor de carga es ejecutado, este tiempo se define en la propiedad "hbase.balancer.period". El objetivo del distribuidor de carga es mover la regiones entre los *HRS* para equilibrar la carga. La ejecución del distribuidor de carga consta de dos pasos: 1) El *HBase master* determina un plan de asignación de regiones dependiendo de la política de balanceo (propiedad "hbase.master.loadbalancer.class") y establece cuáles son las regiones que tiene que ser movidas a cada uno de los *HBaser region servers*. 2) El *HBase master* ejecuta todos los movimientos de regiones necesarios para cumplir con el plan generado.

El distribuidor de carga tiene un tiempo máximo para ser ejecutado, si en ese tiempo no ha terminado la ejecución, ésta es abortada. Este valor puede ser modificado en la propiedad "hbase.balancer.max.balancing" cuyo valor es la mitad del tiempo especificado en la propiedad "hbase.balancer.period". Además, es posible controlar el distribuidor de carga activándolo o desactivándolo mediante la instrucción balance_switch en la shell o utilizando el método de la API de HBase balanceSwitch(). Una vez que el distribuidor de carga ha sido desactivado este no se ejecutará, hasta que no vuelva a ser activado.

HBase ofrece tres políticas de distribución de carga diferentes: 1) *SimpleLoadBalancer*: La más sencilla de las tres, este algoritmo tiene en cuenta el número de regiones que cada *HRS* maneja y la carga que tiene cada uno de ellos. El objetivo es distribuir la carga moviendo las regiones de los *HRSs* de un *HRS* a otro de tal manera que cada uno de ellos tenga el mismo número de regiones.

2) *FavoredNodeLoadBalancer*: Esta política asigna a cada región un conjunto de *HRS* favoritos, de esta manera cada región tiene un *HRS* principal, un secundario y un terciado en donde la región puede ser manejada. El *HRS* especificado como principal es el que se va a encargar de manejar la región, si este *HRS* falla, el secundario será el encargado y pasará a ser nombrado principal.

3) *StochasticLoadBalancer*: Es la política que viene predefinida en la configuración de HBase, la cual decide el mejor *HRS* en el que puede estar una región tratando de minimizar el coste de una función. Esta función se calcula teniendo en cuenta la

carga de la región, la carga de la tabla, la localización de la región y el tamaño de los HFiles, entre otros parámetros que pueden ser configurados por el usuario. Si el coste de la función del estado actual del sistema es menor que el coste de la función de la nueva distribución planeada de las regiones, $F(C) < F(C')$, entonces el proceso de distribución de las regiones no es ejecutado. De esta manera distribuye las regiones entre los *HRS* siempre y cuando el coste de la función calculada con la distribución ideada sea menor al coste actual.

2.6. UPM-CEP

UPM-CEP [16] [17] es un sistema distribuido y paralelo que permite procesar flujos continuos de eventos lo más cerca posible de su origen para poder detectar anomalías o transformarlos antes de ser finalmente almacenados o procesados por otro sistema. UPM-CEP ofrece alta escalabilidad y elasticidad permitiendo ser ejecutado en redes de área extendida. Los eventos (tuplas) son procesados por una serie de operadores conectados entre ellos formando un grafo acíclico. Para distribuir las consultas son divididas en sub-consultas las cuales a su vez pueden ser paralelizadas en varias instancias.

Las tuplas son tipadas y poseen los valores de los campos a procesar. Las tuplas procedentes de una misma fuente tienen que tener el mismo esquema, es decir el mismo número de campos y tienen que ser del mismo tipo. Los campos pueden ser de los siguientes tipos de Java: *timestamp*, *Integer*, *Long*, *Double*, *Byte*, *String*, *Date*.

2.6.1. Operadores

Los operadores se dividen en cuatro grupos: 1) Operadores sin estado: son aquellos que realizan operaciones sobre cada uno de los eventos que llegan al operador; 2) Operadores con estado: son aquellos que realizan las operaciones sobre un conjunto de eventos; 3) Operadores de entrada/salida: son lo operadores encargado de recibir los eventos o enviarlos fuera del sistema y 4) Operadores Definidos por el usuario:

son operadores que pueden ser programados por el usuario. La figura 2.13 muestra la clasificación de los distintos operadores clasificados en los cuatro grupos mencionados.

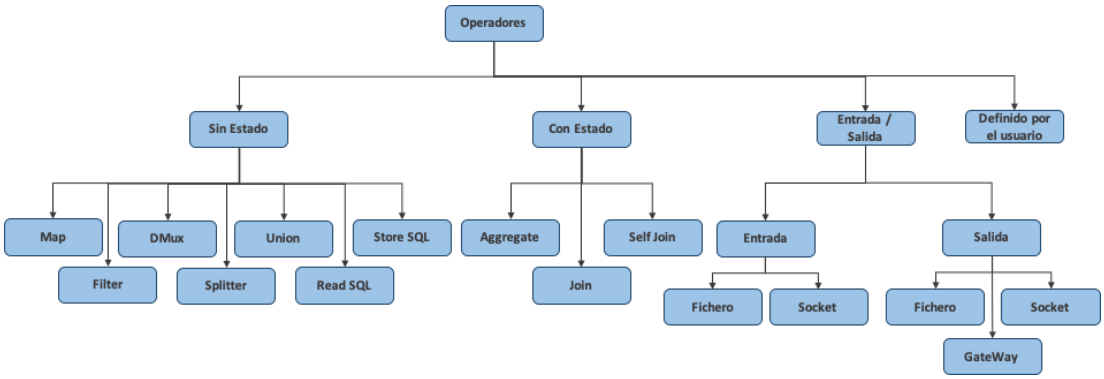


Figura 2.13: UPM CEP operadores clasificados según su funcionalidad.

2.6.1.1. Operadores sin Estado

Los operadores sin estado son aquellos que por cada evento que recibe aplica una serie de modificaciones o comprobaciones sobre los mismos para posteriormente enviarlo por la salida a otro operador. Dentro de este grupo encontramos a los operadores *Map*, *Filter*, *DMux*, *Splitter*, *Union*, *ReadSQL* y *StoreSQL*.

El operador *Map* realiza transformaciones sobre los campos de las tuplas que recibe. Tiene un stream de entrada donde recibe las tuplas, las procesa y las nuevas tuplas generadas son enviadas por un stream de salida al siguiente operador de la consulta. El operador *Filter* comprueba una condición sobre uno o más campos de la tupla recibida, en caso de que se cumpla, la tupla se envía al siguiente operador y en caso contrario se descarta la tupla. El esquema del *stream* de entrada es igual al esquema de los *streams* de salida, ya que no se realiza ninguna transformación sobre las tuplas. El operador *DMux* recibe las tuplas por un stream y pueden ser enviadas por cero, una o más streams según el predicado establecido sobre cada una de las posibles salidas. El esquema de los *streams* de salida tiene que ser el mismo que el del *stream* de entrada.

El operador *Splitter* envía las tuplas que recibe por un stream de entrada por todos los streams de salida que se configuren sin realizar ninguna modificaciones sobre las

tuplas. Por tanto, el esquema del stream de entrada es el mismo que el de todos los streams de salida. Este operador permite multiplicar las tuplas recibidas por el stream de entrada por tantos streams de salida tenga. El operador *Union*, al contrator que el operador *Splitter*, recibe las tuplas por varios streams de entrada y los envía por un stream de salida. El esquema de todos los streams de entrada es igual al esquema del stream de salida.

Los operadores StoreSQL y ReadSQL tienen como finalidad escribir o leer de una base de datos mediante una instrucción SQL respectivamente. Estos operadores necesitan el driver de la base de datos, la conexión JDBC y la instrucción SQL a utilizar. Cuando se configuran estos operadores, antes de comenzar a recibir las tuplas se establece una conexión JDBC con la base de datos indicada. La tabla o las tablas sobre las que se van a realizar las operaciones de escritura o lectura tienen que existir para asegurar el correcto funcionamiento del operador.

2.6.1.2. Operadores con Estado

Los operadores con estado son aquellos que almacenan tuplas en memoria para realizar operaciones (funciones) sobre ellas y generan una o varias tuplas de salida. El conjunto de tuplas almacenadas se denomina ventana. En este grupo están los operadores *aggregate*, *join* y *self-join*. Los operadores que actúan sobre una ventana aplican una operación sobre el conjunto de tuplas almacenadas cuando la ventana se llena. Esto ocurre cuando hay un número determinado de tuplas o cuando ha pasado un tiempo determinado. A continuación se eliminan una o más tuplas de las más antiguas de la ventana (desplazamiento). El número de ventanas utilizadas por el operador se define por la propiedad *groupBy* que tiene cada uno de los operadores con estado disponibles. Esta propiedad *groupBy* permite indicar qué campos de las tuplas que van a ser recibidas va a ser considerados para agrupar las tuplas recibidas por ventana. De esta manera todas las tuplas que tenga el mismo valor en el campo o campos indicado por la propiedad, si no se ha especificado ningún campo el todas la tuplas serán almacenadas en una única ventana.

El operador *Aggregate* agrupa las tuplas en una o varias ventanas, cuando la ventana se desplaza se genera una tupla de salida por cada una de las ventanas con la información obtenida de todas las tuplas almacenadas en cada una de ellas. El operador *Join* obtiene las tuplas desde dos streams de entrada y hace el producto cartesiano con las tuplas almacenadas en las dos ventanas y envía las tuplas generadas tras realizar el producto cartesiano por un stream de salida. El operador *Self-Join* recibe las tuplas desde un stream de entrada, cada vez que recibe una tupla realiza el producto cartesiano entre la tupla recibida y las tuplas almacenadas en las ventanas, las tuplas generadas por el producto cartesiano son enviadas por el stream de salida.

2.6.1.3. Operadores de Entrada y Salida

Los operadores de entrada y salida son aquellos que permiten al UPM-CEP recibir o enviar tuplas fuero del CEP.

Operadores de entrada Los operadores de entrada permiten recibir tuplas desde fuera del UPM-CEP para procesarlas. Hay dos tipos de operadores de entrada, el que recibe tuplas desde un socket y el que lee las tuplas desde fichero. El primer operador de entrada llamado *SocketDataSource*, recibe las tuplas a través de un socket al que se conecta al inicio de la consulta y las envía al siguiente operador de la consulta. Las tuplas recibidas mediante del socket tienen que estar tipadas siguiendo el mismo formato que utiliza internamente el UPM-CEP y El esquema de las tuplas que recibe tiene que ser igual al esquema del stream de salida. El operador *FileDataSource* lee las tuplas desde un fichero, este fichero tiene que poder ser accedido desde el nodo en el que se esté ejecutando este operador, en caso contrario no podrá leer las tuplas. Por cada tupla que lee, la codifica siguiendo el esquema del stream de salida y la envía al siguiente operador de la consulta.

Operadores de salida Los operadores de salida por su parte envían las tuplas ya procesadas desde el CEP por socket, un servicio *REST* o las almacena en un fichero. El operador *SocketDataSink* envía las tuplas recibidas desde el último operador de la consulta a un socket con el cual que establecido previamente la comunicación. Las tuplas enviadas tienen el mismo esquema que el stream de salida del operador desde el

que las recibe ya que no se realiza ninguna modificación sobre las mismas. El operador *GatewayDataSink* envía las tuplas recibidas a una aplicación que tenga un servicio *REST* mediante peticiones *POST*. En caso de que no se puedan enviar las tuplas a la aplicación por un fallo en la conexión con el servicio *REST* las tuplas son almacenadas hasta que se vuelva a establecer la conexión, momento en el que se mandarán a la máxima tasa posible permitida por la red. Si la conexión falla durante un tiempo y el *buffer* donde se almacenan las tuplas se llena el resto de tuplas se van descartando. El operador *FileDataSink* permite escribir las tuplas que recibe desde el operador anterior de la consulta en un fichero. La ruta en la que se va a guardar el fichero tiene existir en el nodo en el que se vaya a ejecutar este operador para asegurar que se va a poder crear el fichero.

2.6.1.4. Operadores Definidos por el Usuario

Los operadores Definidos por el usuario permite al programador definir nuevos operadores. Para ello desde la librería Cliente del UPM-CEP hay que extender la clase AbstractCustomOperator que indica los métodos que hay que implementar definir la funcionalidad del nuevo operador.

2.6.2. Consultas y Sub-Consultas

Las consultas son grafos acíclicos en los que los nodos son los operadores y las aristas son los *streams* de entrada o salida de los operadores. En la figura 2.14 se muestra la representación gráfica de una consulta en la que se pueden observar dos operadores *SocketDataSource* que reciben datos de dos sockets diferentes, cada uno de ellos envían las tuplas a su respectivo operador *Map*. En el caso del operador *Map* de la parte inferior del grafo las tuplas son enviadas a un operador *Filter* y a continuación las tuplas salientes del primer *Map* y del *Filter* se unen en el operador *Union* para ser enviadas al operador *Aggregate*, una vez se desplaza la ventana del operador de genera una nueva tupla que es almacenada en una base de datos mediante el operador

StoreSQL y después las tuplas procesados por la consulta son enviadas a una aplicación REST mediante el operador de salida *GatewayDataSink*.

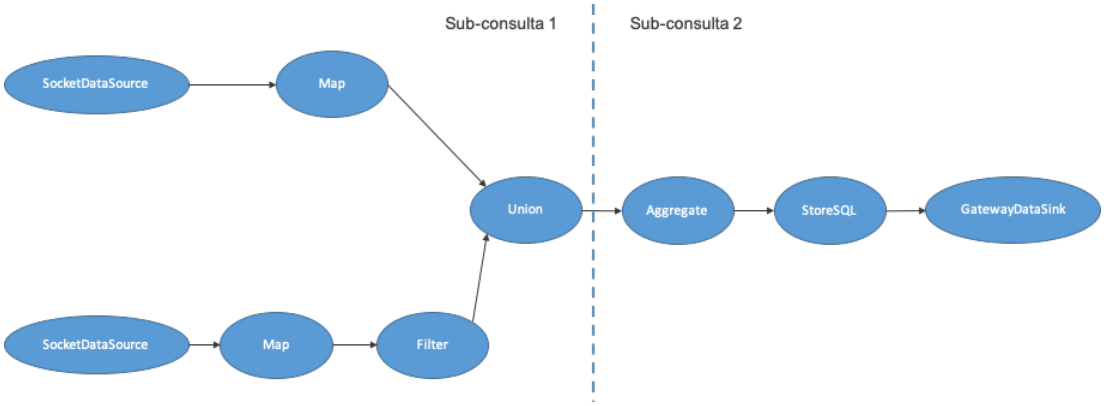


Figura 2.14: Representación gráfica de una consulta.

Una vez el usuario programa la consulta, la envía al CEP para que este la registre. Cuando se registra la consulta esta es dividida en sub-consultas. Una sub-consulta es un conjunto de operadores de los cuales el primero de ellos es un operador con estado excepto en el caso de la primera parte de consulta donde la sub-consulta tiene como primer operado uno o varios operadores de entrada. Por ejemplo en la figura 2.14 se pueden observar dos sub-consultas, la primera, sub-consulta 1, formada por los operadores que están entre los operadores de entrada *SocketDataSource* y hasta el operador *Union* y la segunda sub-consulta, desde el operador *Aggregate* hasta el operador de salida *GatewayDataSink*, sub-consulta 2. Las sub-consultas permiten distribuir y paralelizar las consultas entre distintas máquinas que formen parte del despliegue del UPM-CEP. De esta manera se puede ajustar el número de instancias de cada una de las sub-consultas a la carga recibida permitiendo procesar más o menos tuplas por segundo.

Tras registrar la consulta, el UPM-CEP le indica al usuario las sub-consultas que se han creado y este puede definir el número de instancias que desea desplegar de cada una de las sub-consultas inicialmente y el nodo en le que desea que se despliegue cada una de las instancias de cada sub-consulta.

2.6.3. Balanceadores de carga

Los balanceadores de carga son unos procesos que se ejecutan en cada operadores final de cada sub-consulta. Envía las tuplas generadas por el último operador al primer operador siguiente sub-consulta. Su función es fundamental, ya que permiten distribuir los eventos entre las distintas instancias de las sub-consultas. Hay tres tipos de balanceadores de carga: *Round-Robin*, *Broadcast* y *GroupKey*.

Balanceador Round-Robin

El balanceador Round-Round se instancia de manera automática en todos los operadores, su funcionalidad es distribuir la carga de manera uniforme entre todas las instancias del operador al que envía las tuplas, enviando cada tupla a una instancia de las disponibles de forma ordenada tal y como se muestra en la figura 2.15. En la figura se muestra el balanceador Round-Robin en la parte izquierda y en la parte derecha se pueden ver 3 instancias del mismo operador. Desde el balanceador hasta las instancias del operador se están enviando 6 tuplas, t1, t2, t3, t4, t5 y t6. El balanceador Round-Robin los va a distribuir de tal manera que las tuplas t1 y t4 se envíen a la instancia 1 del operador, las tuplas t2 y t5 a la instancia 2 y el resto, las tuplas t3 y t6 a la instancia 3.

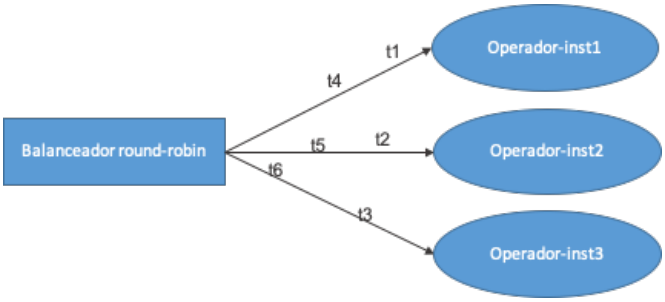


Figura 2.15: Balanceador Round-Robin.

Balanceador Broadcast

El Balanceador Broadcast tiene que ser configurado cuando se definen programa la consulta, es decir el usuario tiene que indicar que desea utilizar este tipo de balanceador en concreto. Este balanceador envía la misma tupla a todas las instancias que haya

del operador al que tiene que enviar. En el ejemplo mostrado en la figura 2.16, se muestra en la parte izquierda el balanceador y en la parte derecha tres instancias del mismo operador. Desde el balanceador a las instancias se envían tres tuplas, t1, t2 y t3, las cuales son enviadas a todas las instancias del operador haciendo que todas las instancias siempre reciban las mismas tuplas.

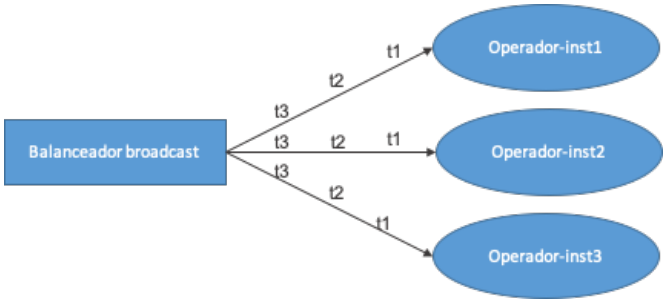


Figura 2.16: Balanceador Broadcast.

Balanceador GroupKey

Finalmente el balanceador GroupKey permite distribuir la carga entre las distintas instancias de un operador teniendo en cuenta valores de uno o varios campos de la tupla. Este balanceador puede ser configurado cuando se especifica el número de instancias a desplegar de las sub-consultas o se configura de manera automática si el operador al que se envían los eventos tiene configurado el campo groupBy. Por ejemplo en el caso de los operadores con estados, se puede configurar la propiedad *groupBy* indicando qué campos van a ser utilizados en la distribución de los eventos.

En la figura 2.17, se muestra el balanceador GroupKey que envía 7 tuplas diferentes a tres instancias del mismo operador. Cuando se instancia el balanceador se crea un array de buckets de 250 posiciones, el tamaño puede ser configurado desde el fichero de configuración del cep, cepconfig.properties, el cual se rellena con los identificadores de las instancias que hay del operador al que hay que enviar los eventos. Siguiendo con el ejemplo, como hay tres instancias y los identificadores son 1, 2 y 3, el array se va a rellenar de la siguiente manera, [1, 2, 3, 1, 2, 3, 1, 2, 3,..., 1, 2, 3]. Cuando el balanceador recibe una tupla, obtiene el valor del campo o campos que han sido especificados en la propiedad *groupBy*, obtiene el código Hash mediante la función

MD5HashFunction de Java y calcula el módulo teniendo en cuenta el tamaño del array de bucket, el módulo es el identificador de la posición del array de buckets que indicara a qué instancia hay que enviar la tupla. De esta manera todas las tuplas cuyos campos especificados en la propiedad *groupBy* sean iguales van a ir a la misma instancia. En el ejemplo de la figura 2.17, se muestra que las tuplas t6 y t7 son enviadas a la instancia 1, las tuplas t1, t2 y t5 a la instancia 2 y las tuplas t3 y t4 a la tercera instancia del operador.

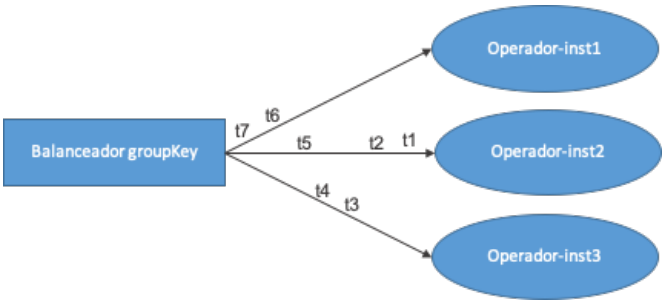


Figura 2.17: Balanceador GroupKey.

2.6.4. Arquitectura

UPM-CEP está compuesto por tres servicios, cep-orchestrator, cep-instanceManager, cep-metricserver y una interfaz Java que permite al usuario interactuar con el servicio cep-orchestrator llamada cep-client, figura 2.18. Además el cep-client tiene todas las clases necesarias para programar las consultas. El servicio cep-orchestrator, es el encargado de gestionar y controlar los distintos cep-instanceManager, registrar, desplegar, eliminar consultas, mover sub-consultas de un cep-instanceManager a otro, incrementar o reducir el número de instancias de las sub-consultas y monitoriza el estado del resto de componentes del sistema. El servicio cep-instanceManager es el encargado de la ejecución de sub-consultas para el procesamiento de las tuplas. Al menos tiene que haber un cep-instanceManager en ejecución en el sistema y pueden desplegarse tantos cep-instanceManagers por máquina como cores haya para aprovechar al máximo los recursos de la máquina. El proceso cep-metricserver, recibe métricas de cada uno de

los componentes que estén en ejecución en el sistema y las exporta a Prometheus [34], un sistema que permite monitorizar cualquier software. ZooKeeper se usa como sistema de almacenamiento persistente del estado del sistema y de las consultas registradas y desplegadas.

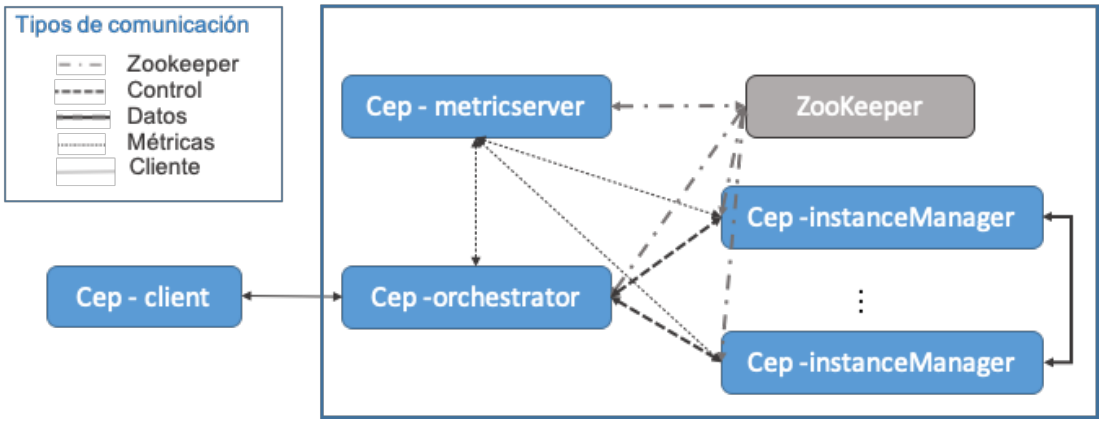


Figura 2.18: Esquema de la arquitectura de UPM-CEP

En la figura 2.18, se muestran las interacciones que se producen entre los distintos servicios del UPM-CEP. Por un lado la interfaz cep-client se comunica únicamente con el cep-orchestrator, el cual sirve las peticiones del usuario y le responde con la resolución de la petición, comunicación cliente. El cep-orchestrator se comunica con el cep-metricserver para informar de las métricas como son el tiempo que tarda en reconfigurar el sistema y el tiempo que tarda en desplegar una sub-consulta, con cada una de las instancias cep-instanceManager para indicar las sub-consultas que tiene que registrar, desplegar o eliminar y con ZooKeeper para almacenar las consultas registradas, desplegadas y el estado del sistema. El servicio cep-instanceManager se comunica con ZooKeeper para registrarse en el sistema, de esta manera el cep-orchestrator sabe que un cep-instanceManager se ha iniciado y lo registra. Los cep-instanceManagers se comunican entre sí para enviar las tuplas que salen procesadas de las subconsultas que están en ejecución en cada cep-instanceManager.

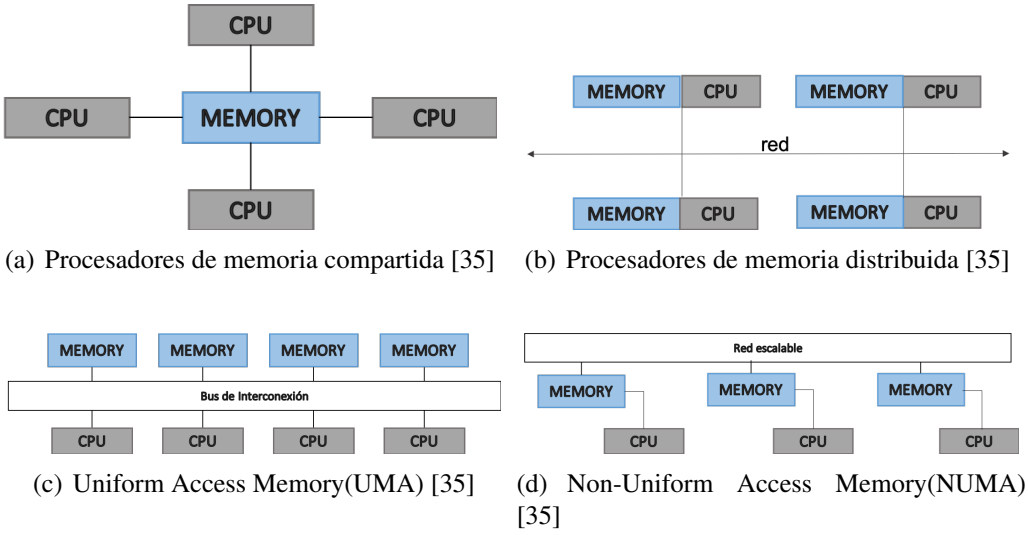
2.7. Arquitecturas de sistemas multiprocesador

Los sistemas multiprocesador [35] son ordenadores con varias CPU o procesadores que comparten memoria y otros componentes como los buses para poder procesar al mismo tiempo diferentes tareas.

Actualmente, se pueden encontrar dos arquitecturas diferentes en los sistemas multiprocesador: La arquitectura de memoria compartida y la arquitectura de memoria distribuida. En las arquitecturas de memoria compartida, los procesadores tienen que compartir el acceso a memoria entre ellos, figura 2.19(a), mientras que en las arquitecturas con memoria distribuida, todos los procesadores tiene su propia memoria local y no tienen un mapa de accesos a memoria entre procesadores. Por ello, para acceder a la memoria en otro procesador se requiere de una comunicación explicita para ello, figura 2.19(b).

Se pueden distinguir dos tipos de arquitecturas dentro de los sistemas con arquitectura de memoria compartida teniendo en cuenta cómo es el acceso a memoria entre los procesadores. 1) Arquitectura de acceso uniforme a la memoria, Uniform Access Memory Architecture (UMA), y 2) Arquitectura de acceso no uniforme a la memoria, Non-Uniform Access Memory Architecture (NUMA) [36].

Las arquitecturas UMA están compuestas por un conjunto de procesadores o CPUs idénticos donde el acceso a todas las regiones disponibles de memoria es igual. El acceso a memoria desde un procesador se realiza mediante un bus de interconexión, figura 2.19(c). Por otro lado, las arquitecturas NUMA contiene un conjunto de procesadores los cuales tiene directamente conectada una o varias tarjetas de memoria RAM y todos ellos están conectados a través de una red escalable, figura 2.19(d). La principal diferencia entre las arquitecturas UMA y NUMA es el coste de acceso a las diferentes regiones de memoria, mientras que en las arquitecturas UMA el coste del acceso es el mismo, en las arquitecturas NUMA el acceso a la memoria directamente conectada al procesador es más rápido que el coste de acceder a la memoria conectada a otra CPU.



(a) Procesadores de memoria compartida [35] (b) Procesadores de memoria distribuida [35]

(c) Uniform Access Memory(UMA) [35] (d) Non-Uniform Access Memory(NUMA) [35]

Figura 2.19: Procesadores de memoria compartida

2.7.1. Non-Uniform Memory Access (NUMA)

Los sistemas NUMA permiten asociar a los procesos memoria y el procesador en el que va a ejecutarse. La figura 2.20 muestra la distribución de la memoria entre los distintos procesadores de una maquina con procesadores AMD Opteron 6376 @ 2.3GHz y 128 GB de memoria RAM y la figura 2.21 muestra la ampliación de la distribución de memoria del Socket 1 de la misma máquina. En total son 64 cores virtuales distribuidos en 4 sockets, cada socket tiene 2 unidades NUMA y cada unidad NUMA contiene 8 cores. Cada core tiene directamente conectada la cache L1, cada dos cores comparten la cache L2 y cada 8 cores comparten la cache L3. Todos los cores de una unidad NUMA comparten el acceso a la región de memoria que se encuentra directamente conectada al socket.

Cuando un proceso es lanzado, este comienza a ejecutar en un core elegido aleatoriamente y en caso de consumir memoria RAM, este comienza a utilizar la memoria directamente conectada con el socket en el que se encuentra el core en el que está ejecutándose para tener la mínima latencia en el acceso a la memoria. Periódicamente, Linux analiza si tiene que distribuir el consumo de CPU entre los distintos sockets y en caso de que sea necesario el proceso se mueve de su core a otro. Primero se intenta

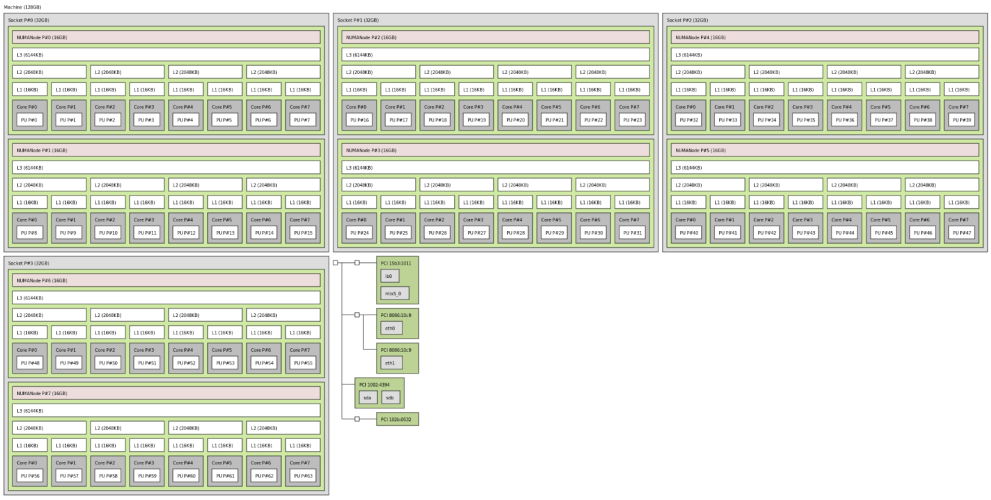


Figura 2.20: Distribución NUMA y memoria en una máquina AMD Opteron 6376 @ 2.3GHz

mover al core que comparte la caché L2 con el core en el que está ejecutando, si no es posible porque está ocupado se intenta mover a otro core que comparta la caché L3 y si esto no es posible el proceso se mueve a otra unidad NUMA. Si esto último ocurre, se producirá una degradación en el rendimiento ya que el proceso está ejecutándose en una unidad NUMA diferente a la unidad NUMA en la que tiene reservada la memoria y por lo tanto el acceso a la misma tiene una latencia extra.

El sistema operativo Linux tiene diferentes herramientas que permiten trabajar con la arquitectura NUMA. Por ejemplo el comando ”numactl –hardware”muestra la configuración NUMA y la tabla de latencias llamada System Locality Information Table (SLIS), en ella se representa la distancia de acceso desde un nodo a las regiones de memoria de otros nodos. La figura 2.22 muestra la salida del comando numactl en la misma máquina AMD y a la izquierda aparece una representación del esquema completo de la arquitectura hardware. Cada una de las 8 unidades NUMA está formada por 8 cores. La unidad NUMA 0 tiene las CPUs 0 a 7, la unidad NUMA 1 las CPUs 8 a 15 y así sucesivamente. Cada unidad NUMA tiene directamente conectada dos ranuras con tarjetas de 8 GB de RAM cada una (16 GB de memora RAM en total). La máquina tiene directamente conectados dos discos en la unidad NUMA 0.

Figura 2.21: Distribución de la memoria en el Socket 1 de una máquina AMD Opteron 6376 @ 2.3GHz

Observando la tabla de latencias (SLIT), desde la unidad NUMA 0 se puede acceder a la memoria directamente conectada con distancia 10. Si el proceso corriendo en esa unidad NUMA tiene que acceder a la memoria conectada a otra unidad conectada directamente mediante un bus, el coste de ese acceso será de 16, por ejemplo un proceso en la unidad NUMA 0 tiene que acceder a la memoria de la unidad NUMA 6. Pero si el proceso tiene que acceder a la memoria de una unidad NUMA que no está conectada directamente mediante el bus, el coste es 22, por ejemplo acceder a la memoria de la unidad NUMA 3.

Asignar a un proceso la unidad NUMA o conjunto de cores en concreto donde va a estar siempre ejecutando permite asegurar que el proceso siempre accederá en primer lugar a la memoria directamente conectada a la unidad NUMA y no va a ser desplazado a otra unidad NUMA durante la ejecución evitando latencia extras al tener que acceder a regiones de memoria que se encuentran en otras unidades NUMA. Para ello se utiliza el comando “numactl –physcpubind= $CPUs –localalloc” antes de la

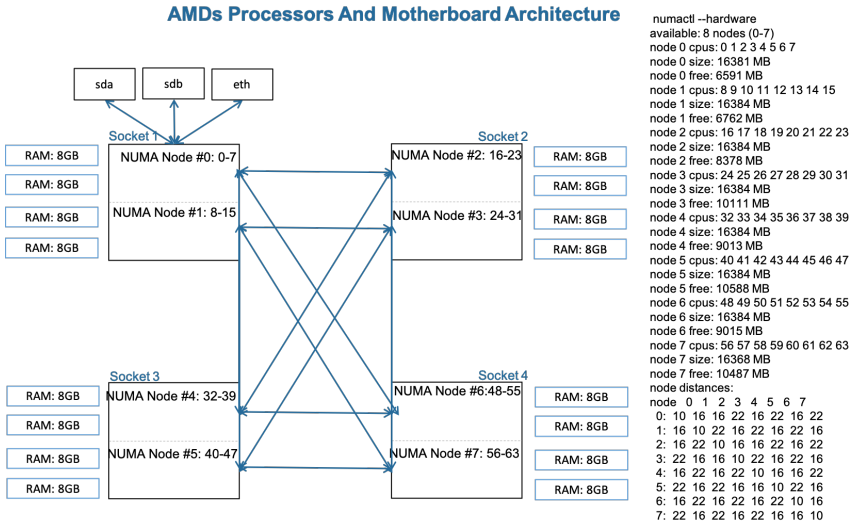


Figura 2.22: Arquitectura NUMA de una máquina AMD Opteron 6376 @ 2.3GHz

llamada a la ejecución del proceso, donde $CPUs son los cores a los que se va a asociar el proceso y con la opción –localalloc se indica que utilice la memoria directamente conectada.

Otra herramienta muy útil es el comando ”taskset”que permite conocer a qué core está asociado un proceso en concreto. Si no se ha vinculado el proceso a ninguna CPU la salida del comando “taskset -c -p $PROCESS_ID”, donde $PROCESS_ID es el identificador del proceso, será el rango de todos los cores (0-36). En cambio, si se ha asociado el proceso a un core o conjunto de cores por ejemplo del core 8 al 15, la salida del comando será 8-15.

Capítulo 3

Inyector de Datos Masivo

Hoy en día se generan grandes cantidades de datos a través por ejemplo de las redes sociales, compras y visitas a sitios web. Estas datos son procesados y analizados posteriormente con distintas herramientas. Previo al análisis de los datos, estos tienen que ser cargados en un sistema que permita su análisis. Dependiendo de la cantidad de datos, este proceso puede llevar varias horas o incluso días. Por este motivo se ha desarrollado un inyector de datos masivos para HBase.

El objetivo es minimizar el tiempo de carga empleado de manera eficiente los recursos del sistema y al mismo tiempo realizar una distribución uniforme entre las máquinas del sistema.

En este capítulo se va explicar en qué consiste el proceso de carga masiva de datos, parte del proceso de extracción, transformación y carga en una base de datos (sección 3.1). Este proceso, conocido por sus siglas en inglés ETL (Extract-Transform-Load), permite mover datos desde distintas fuentes, realizar modificaciones sobre los mismos y finalmente almacenarlos en una base de datos. El proceso de carga de datos se va a realizar sobre HBase, una base de datos NoSQL de tipo clave valor presentada en el capítulo 2. HBase tiene dos métodos para insertar datos en las tablas previamente creadas, Put e ImportTSV [14] los cuales han sido analizados en la sección 3.2.

En las siguientes secciones se presentan las principales contribuciones de este capítulo. Una herramienta basada en ImportTSV, el inyector de carga, que permite realizar la carga de datos en HBase cuando la clave que identifica a la fila de la ta-

bla a cargar se compone de varias columnas, se presenta en la sección 3.4. Con esta herramienta se puede cargar datos de tal manera que las claves generadas permiten realizar consultas sobre los datos insertados de manera más eficiente. Esta herramienta que permite paralelizar el proceso de carga para disminuir el tiempo de carga. En la sección 3.5, se muestra un algoritmo, que posteriormente ha sido implementado, para poder dividir las tablas de HBase sin tener conocimiento de los datos que se van a cargar y distribuirlas entre los distintos nodos del sistema, paralelizando el proceso de carga. Esta herramienta, puede ser utilizada tanto con la herramienta ImportTSV como en su versión modificada, el inyector de carga.

Ambas herramientas, ImportTSV e Inyector de carga, utilizan MapReduce para llevar a cabo su función. La configuración de los recursos de todos los servicios utilizados durante el proceso de carga: HDFS, ZooKeeper, HBase y MapReduce (presentados en el capítulo 2) tienen un papel fundamental para maximizar el uso de los recursos que proporcionan las distintas máquinas del sistema. Debido a la gran diversidad de máquinas que podemos encontrar hoy en día, se ha desarrollado una herramienta la cual teniendo en cuenta los recursos de cada una de las máquinas del sistema independientemente de si el sistema ha sido desplegado en un clúster homogéneo como heterogéneo, distribuye dichos recursos entre todos los servicios y permite la configuración más adecuada para cada uno de ellos. Esta herramienta se presenta en detalle en la Sección 3.6.

Las principales contribuciones de este capítulo son: 1) el inyector de carga que permite insertar de datos de forma masiva generando claves mediante la concatenación de los valores de varias columnas de la fila; 2) Pre-Split paralelizar el proceso de carga creando las regiones uniformes de las tablas antes de iniciar la carga. 3) Distribución de recursos, una herramienta que permite distribuir los recursos de cada una de las máquinas de sistema entre los servicios que se vayan a desplegar en cada una de ellas. 4) Configuración de los recursos asignados a YARN, esta herramienta permite distribuir los recursos de tal manera que se consigue el mayor paralelismo posible durante el proceso de carga aprovechando al máximo los recursos. 5) Estudio del impacto del tamaño de bloque de HDFS en el rendimiento en el proceso de carga de datos en

HBase. 6) Ubicación de regiones, una herramienta que permite almacenar las localizaciones de los datos en cada una de las máquinas de sistema una vez que ha terminado el proceso de carga. 7) Una evaluación de cada una de la contribuciones anteriores en tres despliegues distintos demostrando el impacto de las mismas durante el proceso de carga.

3.1. Extracción, Transformación y Carga (ETL)

El concepto de ETL [37], se popularizó en los años 70 cuando las empresas comenzaron a integrar la información que hasta entonces habían estado almacenando en distintos repositorios de datos con el fin de almacenarlos en uno solo. De ahí surgió la necesidad de "Extraer" los datos de cada uno de esos repositorios, "Transformarlos" para modificar su formato, limpiar los datos y finalmente "Cargarlos" en el nuevo repositorio.

3.2. Mecanismos de carga de datos en HBase

HBase proporciona dos métodos para almacenar datos en una tabla. El más sencillo es el método *Put* el cual añade los datos en una única fila de la tabla. En el Código 3.1, se muestra un programa java que inserta datos en una tabla llamada "sensor" empleando este método. En primer lugar hay que crear el objeto *HTable* que va a permitir ejecutar modificaciones y lecturas sobre la tabla "sensor" (línea 5, código 3.1). A continuación, cada vez que se desee realizar una inserción de datos en la tabla hay que crear una instancia del objeto *Put*, pudiendo agrupar modificaciones sobre la misma fila de la tabla. Este objeto *Put* puede reutilizarse para añadir otras filas una vez se haya completado el proceso de carga de la actual. Por ejemplo, supongamos que se van a insertar datos en la tabla "sensor" y que dicha tabla tiene dos familias de columnas ("temp" y "pres"), para insertar datos en la fila identificada mediante la clave "claveSensor" hay que instanciar un objeto *Put* con la clave de la fila (línea 7, código 3.1),

a continuación por cada valor a insertar se añade al objeto *Put* el valor deseado indicando la familia de columnas, la columna, la marca de tiempo de la inserción (ts) y el valor. Finalmente, se llama al método *put* del objeto *HTable*, el cual permite insertar los datos almacenados en el objeto *Put*.

```
1  public class EjemploPut {
2    ...
3     public void putUnaFila() {
4       ....
5       HTable hTable = new HTable(config,"sensor");
6       long ts = System.currentTimeMillis();
7         Put put = new Put(Bytes.toBytes(claveSensor));
8         put.add(Bytes.toBytes("info"), Bytes.toBytes("id"), ts,
              IdSensor);
9         put.add(Bytes.toBytes("temp"), Bytes.toBytes("in"), ts,
              valorTIn);
10        put.add(Bytes.toBytes("temp"), Bytes.toBytes("out"),
              ts, valorTOut);
11        put.add(Bytes.toBytes("pres"), Bytes.toBytes("in"), ts,
              valorPIn);
12        put.add(Bytes.toBytes("pres"), Bytes.toBytes("out"),
              ts, valorPOut);
13         hTable.put(put);
14     }
15     ....
16 }
```

Código 3.1: Ejemplo código JAVA método Put

El segúndo método de carga dispobible en HBase es ImportTSV [14]. Esta herramienta permite importar datos desde un fichero a una tabla HBase, utilizando HDFS y MapReduce. Para la ejecución de la herramienta la tabla en la que se van a cargar los datos tiene que estar creada y el fichero tiene que estar almacenado en HDFS. ImportTSV lanza un trabajo MapReduce el cual va leyendo una a una las líneas del fichero almacenado en HDFS y crea los ficheros de datos de la tabla (HFiles). Este método requiere que el fichero tenga cierta estructura y es que una de las columnas tiene que ser la clave de la fila, esa columna no va ha formar parte de las columnas de la tabla. Mediante el proceso *map* del trabajo MapReduce, se procesan cada una de las líneas

del fichero creando por cada una de ellas un objeto Put del API Java de HBase y el proceso *reduce* escribe el contenido de los objetos Put ordenados lexicográficamente teniendo en cuenta la clave en cada uno de los ficheros HFiles que tenga la tabla.

La herramienta requiere de cierta información la cual se indica mediante los parámetros -Dimporttsv.separator y -Dimporttsv.columns además del nombre de la tabla y la localización del fichero almacenado en HDFS. El parámetro *-Dimporttsv.separator* indica el delimitador de campo, es decir, cada uno de los campos de cada fila del fichero está separado por dicho delimitador. El parámetro *-Dimporttsv.columns* especifica en qué orden se encuentran cada uno de los campos de la tabla, se compone de una cadena separada por comas de tal manera que cada nombre de campo se separa por "," y cada par familia de columnas - columna se separa por ":". ImportTsv se lanza mediante el siguiente comando: "hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -Dimporttsv.separator=, -Dimporttsv.columns="HBASE_ROW_KEY,info:id,temp:in, temp:out,pres:in,pres:out" sensor hdfs://tmp/hbase.csv". En este caso el delimitador de los campos es ",". Y los campos de la tabla se especifican de esta manera el primer campo de la línea a procesar contiene la clave de la fila, el segundo campo es el valor de la columna "info:id", el tercer y cuarto campo contiene el valor "in" y "out" de la familia de columnas "temp" y los campos 5 y 6 contienen los valores de las columnas "in" y "out" de la familia de columnas "pres".

Si el fichero que se va a cargar no tiene una columna clave esta herramienta no puede ser utilizada, a no ser que se modifique el fichero y se incluya dicha columna para identificar de manera única cada una de las filas del fichero.

La principal diferencia entre ambos métodos es que con el método Put cada una de las filas de la tabla es insertada a través de la API Java de HBase, la cual escribe el dato en el fichero HFile de HDFS correspondiente y el método ImportTSV escribe directamente los datos en el fichero HFile almacenado en el directorio MapReduce de HDFS sin interactuar con HBase y tras finalizar el trabajo MapReduce el fichero HFile es volcado en el directorio de HBase dentro de HDFS. Además, con el método ImportTSV parte del proceso se paraleliza ya que por cada bloque en el que es dividido el fichero,

al ser almacenado en HDFS, se ejecuta una tarea *map*, lo que permite que varias tareas sean ejecutadas al mismo tiempo durante la fase *map* del proceso MapReduce.

3.2.1. HBase durante el proceso de carga

Cuando se crea una tabla en HBase, solo tiene una región. Una región es un conjunto de filas ordenadas por la clave que identifica unívocamente a cada una de las filas de la tabla. Cada región se identifica por la clave de inicio y la clave final del reango de claves que pueden pertenecer a la misma.Cuando la tabla está vacía HBase no conoce los datos que se van a insertar en la misma y por ello solo se crea una región cuyos límites son vacíos, "[,)".

En la figura 3.1 se muestra como actúa HBase durante el proceso de carga, inicialmente los datos que se van insertando se almacenan en la región inicial, Región 1 en la figura. A medida que se van añadiendo datos, el tamaño de la región aumenta, HBase antes de añadir los datos en la región comprueba si el tamaño actual de la región, por ejemplo 9,8 GB, más el tamaño de los datos a insertar, 300 MB, no supera un tamaño definido por el usuario y que está preconfigurado en HBase a 10 GB. Al superarse el tamaño, HBase no añade los datos y comienza con el proceso de partición de la región o *split* (paso 1 de la figura 3.1). Primero HBase crea dos nuevas regiones, Región 1-1 y Región 1-2, para ello utiliza uno de los algoritmos de *split* que tiene disponibles. Estos algoritmos están definidos en el Capítulo 2, los cuales mediante diferentes estrategias calculan la clave por la que se va a dividir la región en dos nuevas regiones. Las claves que definen a la Región 1-1 son "[, *N*)" y los que definen a la Región 1-2 son "[*N*,)". Tras este proceso comienza a copiar los datos de las filas cuyas claves van desde la primera clave de la Región 1, por ejemplo *A*, hasta la anterior clave por la que se ha dividido la Región 1 en 2, *N-1* y desde la *N* hasta el final en la Región 1-2 (paso 2 de la figura 3.1). Una vez se han copiado los datos, la región original (Región 1) se elimina y las dos nuevas regiones, Región 1-1 y Región 1-2, comienzan a recibir los datos a almacenar. En este punto los datos que iban a ser añadidos en la Región 1, se añaden en la Región 1-1 o en la Región 1-2 dependiendo de la clave de la fila a añadir (paso

3 de la figura 3.1). Este proceso de creación de nuevas regiones hace que el proceso de carga de datos se vea detenido mientras dura la partición de regiones, ya que la modificación de la región paraleliza la inserción de nuevos datos.

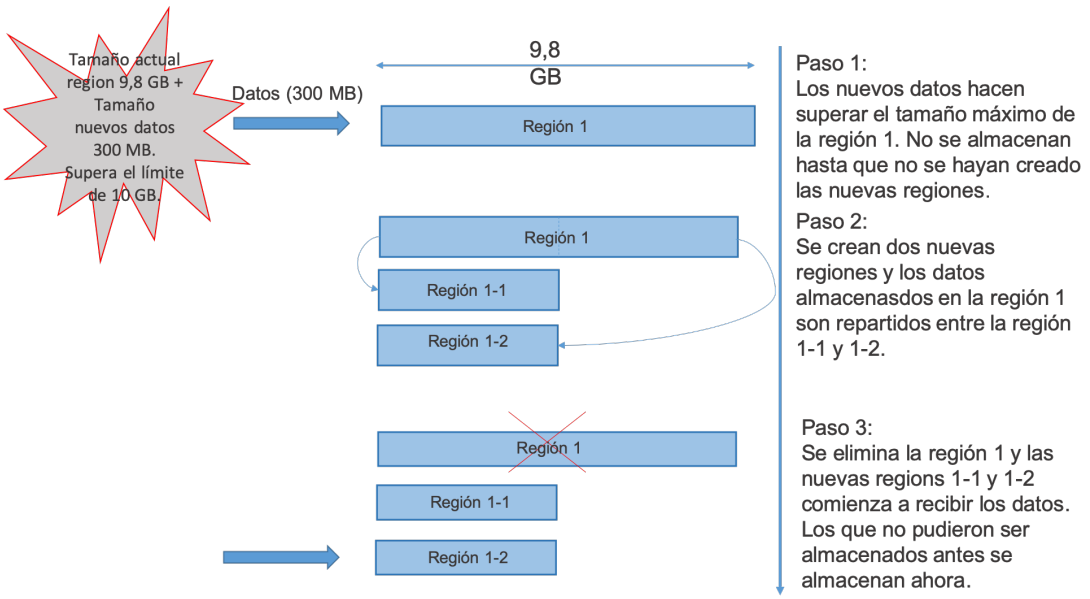


Figura 3.1: Creación de nuevas regiones.

En el caso de utilizar el método de carga ImportTSV, HBase no conoce las claves de los datos que va a tener que servir hasta que finaliza el proceso MapReduce. Por ello, debido a que la tabla al crearse consta de una única región todos los datos son almacenados en dicha región durante el proceso MapReduce, a pesar de que el tamaño supere los límites predefinidos en HBase. Una vez ha terminado la ejecución de ImportTSV, cuando HBase comienza a servir los datos y detecta que la región que está sirviendo supera los límites predefinidos, HBase lanza el proceso de *split* para crear las correspondientes regiones siguiendo el mismo procedimiento que cuando se están añadiendo datos de manera continua. En este caso, el proceso de carga no se ve interrumpido ya que ya ha finalizado, pero durante la creación de las nuevas regiones la tabla deja de estar operativa al tener que copiar los datos de la región a dividir en las dos nuevas regiones.

3.3. Configuración de los recursos YARN

3.3.0.1. Configuración predefinida de YARN

YARN por omisión tiene definidos unos valores de configuración para algunas de las propiedades(tabla 3.1). Estos no tienen encuenta las características de la máquina en la que va a ser ejecutado. Para cada uno de los procesos se define la memoria, la memoria para la JVM y la cantidad de cores virtuales. Los recursos asignados al servicio NodeManager son 8192 MB de memoria RAM y 8 cores virtuales (vcores), definidos en las propiedades yarn.nodemanager.resource.memory-mb y yarn.nodemanager.resource.cpu-vcores, respectivamente. Las propiedades yarn.scheduler .minimum-allocation-mb y yarn.sche-duler.maximum-allocation-mb especifican la cantidad mínima y máxima de memoria que puede ser asignada a cada contenedor, establecidas a 1024 MB y 8192 MB de memoria RAM respectivamente. Por ejemplo, para el proceso *Application Master* la cantidad de memoria es de 1536 MB de RAM de los cuales 1024 MB van a ser utilizados por la JVM y 1 vcore (yarn.app.mapreduce.am.resource.mb, yarn.app.mapreduce.am.command-opts y yarn.app.mapreduce.am.resouce.cpu-vcores). Los contenedores *map* son configurados con 1024 MB de RAM (mapreduce.map.memory.mb), 819 MB de memoria RAM (mapreduce.map.java.opts) y 1 vcore (mapreduce.map.cpu.virtual). Los mismos valores son definidos para los contenedores *reduce* en sus respectivas propiedades: mapreduce.reduce.memory.mb, mapreduce.reduce.java.opts y mapreduce.reduce.cpu.virtual.

La figura 3.2 muestra la distribución de recursos de YARN utilizando esta configuración predefinida en un máquina que tenga al menos 8192 MB de RAM y 8 cores virtuales, si tiene menos capacidad no será posible desplegar el servicio YARN NodeManager. En la figura 3.2 se diferencian las dos situaciones que pueden darse en una de las máquinas de un clúster con al menos dos máquinas, en ambos casos se representa un total de 8192 MB y 8 vcores los cuales son distribuidos en contenedores representados por los rectángulos azules (claro y oscuro). En la parte superior se muestra el caso en el que el contenedor del proceso *Application Master* ha sido desplegado (rectángulo azul claro) y en la parte inferior cuando el contenedor *Application Master* ha sido

Tabla 3.1: Configuración predefinida de YARN [5]

Propiedad	Valor
yarn.scheduler.minimum-allocation-mb	1024
yarn.scheduler.maximum-allocation-mb	8192
yarn.nodemanager.resource.memory-mb	8192
yarn.nodemanager.resource.cpu-vcores	8
mapreduce.map.memory.mb	1024
mapreduce.map.java.opts	-Xmx819m
mapreduce.map.cpu.virtual	1
mapreduce.reduce.memory.mb	1024
mapreduce.reduce.java.opts	-Xmx819m
mapreduce.reduce.cpu.virtual	1
yarn.app.mapreduce.am.resource.mb	1536
yarn.app.mapreduce.am.command-opts	-Xmx1024m
yarn.app.mapreduce.am.resouce.cpu-vcores	1

desplegado en otra máquina del clúster. El resto de rectángulos azules oscuro indican contenedores que pueden ser utilizados tanto por tareas *map* como *reduce*. La distribución de los recursos en contenedores permiten conocer cuantos contenedores pueden ejecutarse en paralelo con la configuración predeterminada de YARN en ambos casos.

En el caso en el que el contenedor del proceso *Application Master* ha sido desplegado en la máquina, este ha ocupado 1536 MB y 1 vcore de los 8192 MB y 8 vcores de la configuración predefinina que tiene disponibles el servicio YARN_NM para distribuir entre los contenedores (rectángulo azul claro). El resto 6656 MB y 7 vcores es repartido en contenedores de tamaño 1024 MB y 1 vcore (rectángulos azul oscuro) dando lugar a la posibilidad de ejecutar en paralelo hasta 6 contenedores *map* o *reduce* quedando sin asignarse 512 MB y 1 vcore ya que no se puede crear un contenedor con solo 512 MB de memoria RAM (rectángulos con 512 MB y 1 vcore) y por lo tanto esos recursos no serán utilizados en ningún momento de la ejecución del trabajo MapReduce. Por otro lado, en el caso en el que el contenedor del proceso *Application Master* no ha sido desplegado en esa máquina (parte inferior de la figura 3.2), los 8192 MB y 8 vcores disponibles por el servicio YARN NodeManager para crear los contenedores permite crear hasta 8 contenedores con 1024 MB y 1 vcore cada uno, rectángulo azul oscuro.

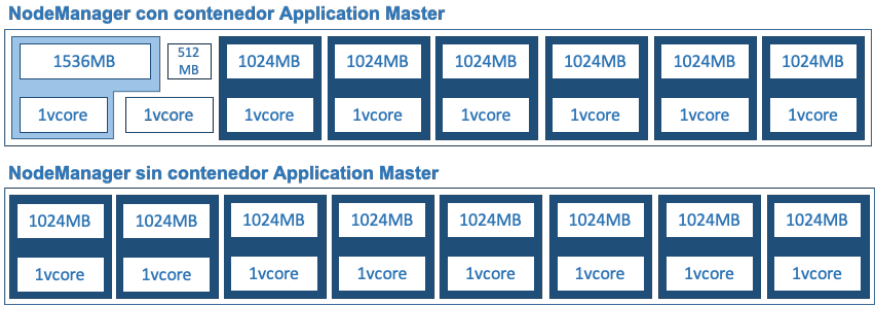


Figura 3.2: Distribución de contenedores con la configuración predefinida de YARN.

La configuración predefinida de YARN puede ser de utilidad en máquinas que tengan 8 cores virtuales y 8192 MB de RAM, pero si la máquina tiene más recursos estos no serán utilizados y por lo tanto no se están aprovechando al máximo la capacidad de procesamiento de la máquina.

3.3.0.2. Configuración de YARN utilizando *yarn-util.py*

Para configurar YARN teniendo en cuenta las características de las máquinas, HortonWorks desarrolló *yarn-util.py* [38] que indicando el número de cores virtuales, el tamaño de la memoria RAM, el número de discos e indicando si va a estar HBase corriendo en la misma máquina, devuelve los valores con los que hay que configurar las propiedades de memoria RAM de YARN (tabla 3.2). Con estos parámetros el programa realiza los siguientes cálculos: 1) La cantidad de memoria que puede utilizar YARN teniendo en cuenta la memoria utilizada por el sistema operativo y la memoria que usará HBase, en caso de encontrarse ejecutando en la misma máquina. 2) El número de contenedores que pueden ejecutarse en paralelo siendo el mínimo de tres valores: a) dos veces el número de cores, b) 1,8 veces el número de discos y c) la cantidad total de memoria RAM disponible dividida entre el mínimo tamaño de los contenedores. Este último valor, el mínimo tamaño de los contenedores se obtiene de una tabla disponible en [38]. 3) La cantidad de memoria RAM de cada contenedor: siendo el máximo de el tamaño mínimo de los contenedores y la cantidad de memoria RAM disponible entre el número de contenedores. Con esto calcula los valores de las distintas propiedades

Tabla 3.2: Propiedades de la configuración de HortonWorks para YARN

Propiedad	Valor
yarn.scheduler.minimum-allocation-mb	RAM-por-Contenedor
yarn.scheduler.maximum-allocation-mb	#Contenedores * RAM-por-Contenedor
yarn.nodemanager.resource.memory-mb	#Contenedores * RAM-por-Contenedor
mapreduce.map.memory.mb	RAM-por-Contenedor
mapreduce.map.java.opts	0,8 * RAM-por-Contenedor
mapreduce.reduce.memory.mb	2 * RAM-por-Contenedor
mapreduce.reduce.java.opts	2 * 0,8 * RAM-por-Contenedor
yarn.app.mapreduce.am.resource.mb	2 * RAM-por-Contenedor
yarn.app.mapreduce.am.command-opts	2 * 0,8 * RAM-por-Contenedor

de los ficheros de configuración de YARN yarn-site.xml y mapred-site.xml, tal y como se muestra en la tabla 3.2.

El resto de propiedades no se modifican y se puede ver sus valores predefinidos en [5], las que corresponden con el número de cores virtuales asignados en total al YARN_NM y a los distintos tipos de contenedores son 8 vcores y 1 vcore, respectivamente. Esto significa que el número máximo de contenedores es 8 independientemente de la configuración del resto de propiedades. Debido a que el número de cores virtuales que se asignan al YARN_NM es de 8 y a cada uno de los contenedores que se van a crear se va a asignar 1 vcore, el máximo número de contenedores que se van a crear es 8.

En la figura 3.3, se muestra la salida proporciona tras la ejecución de *yarn-utils.py* para una máquina con 64 vcores, 131072 MB de memoria RAM, 2 discos y con HBase ejecutándose en ella. Como se puede observar, en la salida no se especifica la distribución que hay que hacer de los cores virtuales entre los distintos tipos de contenedores, por lo que la configuración se las propiedades asignadas a la distribución de los cores virtuales es la misma que la configuración predeterminada de YARN. La cantidad total de memoria RAM que se asigna al servicio YARN_NM es de 81920 MB (yarn.scheduler.maximum-allocation-mb) y la mínima cantidad de memoria RAM que se puede asignar a un contenedor es 20480 MB (yarn.scheduler.minimum-allocation-mb). Con esta configuración en esta máquina tendríamos la posibilidad de ejecutar en paralelo un máximo de 4 contenedores ya que la cantidad de memora establecida para

```
$ python yarn-utils.py -c 64 -m 128 -d 2 -k True
 Using cores=64 memory=128GB disks=2 hbase=True
 Profile: cores=64 memory=81920MB reserved=48GB usableMem=80GB disks=2
 Num Container=4
 Container Ram=20480MB
 Used Ram=80GB
 Unused Ram=48GB
 yarn.scheduler.minimum-allocation-mb=20480
 yarn.scheduler.maximum-allocation-mb=81920
 yarn.nodemanager.resource.memory-mb=81920
 mapreduce.map.memory.mb=20480
 mapreduce.map.java.opts=-Xmx16384m
 mapreduce.reduce.memory.mb=20480
 mapreduce.reduce.java.opts=-Xmx16384m
 yarn.app.mapreduce.am.resource.mb=20480
 yarn.app.mapreduce.am.command-opts=-Xmx16384m
 mapreduce.task.io.sort.mb=8192
```

Figura 3.3: Salida de *yarn-util.py* para una máquina con 64 vcores, 128 GB de RAM, 2 discos y con HBase ejecutándose

cada contenedor sin importar el proceso que vaya a ejecutarse en él, es de 20480 MB llegando a consumir toda la memoria asignada al servicio YARN NodeManager con 4 contenedores.

En cuanto a la cantidad de memoria RAM asignada a los contenedores *map*, que siguiendo con el ejemplo de la figura 3.3 se ha establecido a 20480 MB, ésta puede ser mucho más grande que el tamaño del bloque del fichero almacenado en HDFS que va a ser procesador por la tarea, produciendo que gran parte de la memoria no sea utilizada. La cantidad de memoria ideal para los contenedores *map* debe de ser el doble del tamaño del bloque del fichero a procesar, ya que durante la ejecución de las tareas *map*, el bloque del fichero almacenado en HDFS que se va a procesar se guarda en memoria si hay suficiente espacio, si no se va leyendo según se va procesando. Además los pares clave-valor generados cómo salida de la tarea *map* también son almacenados en memoria y en caso de que no haya suficiente espacio se escriben a disco. De ahí que si el bloque del fichero es de 128 MB, durante la ejecución de la tarea *map* se consumirán aproximadamente 256 MB, por lo tanto con la asignación de 20480 MB al

contenedor, 20224 MB se quedarían sin utilizar.

3.4. Inyector de carga

En muchas ocasiones las claves de las filas se tratan de un identificador numérico único, como puede ser un valor incremental. En HBase se accede al dato deseado a través de la clave, de ahí la importancia de generar claves que permitan ejecutar consultas de manera eficiente. Para ello, en primer lugar hay que observar las consultas más frecuentes a realizar sobre la tabla y definir cuales son los campos más utilizados en los filtros de las consultas. De esta manera, se puede considerar que los todos o parte de los campos utilizados en los filtros formen parte de la clave.

Por ejemplo, el fichero que contiene los datos tiene una columna con un valor incremental el cual es considerado como clave, además tiene los valores de otras columnas más. Observando las consultas a realizar, en la mayoría de ellas se utilizan los campos 2 y 5 para filtrar. Si, las filas se añaden utilizando como clave el valor incremental, cada vez que se hace una consulta hay que recorrer todas las filas y filtrar por los campos que se indiquen. En cambio, si se añaden estos campos a la clave de tal manera que los valores se concatenen, por ejemplo "valorCampo2valorCampo5incremental" al hacer la consulta solo se recorrerán las filas cuya clave tenga los valores de los campos 2 y 5.

La herramienta ImportTSV que encontramos disponible en HBase, no permite realizar esta combinación de columnas para crear la clave, ya que ésta tiene que ser una de las columnas del fichero. Para utilizar esta herramienta en este caso se podría preprocesar el fichero y crear para cada una de las filas la clave deseada. Esta tarea puede llegar a ser muy costosa cuando el fichero a preprocesar contiene millones de filas.

Para evitar este preprocesamiento y crear claves de tal manera que permitan ejecutar las consultas de manera más eficiente se ha implementado una herramienta basada en ImportTSV, que al igual que ella procesa los ficheros almacenados previamente en HDFS y mediante un proceso MapReduce añade las filas a las regiones de HBase que más tarde servirán los datos. La principal aportación de esta herramienta es la creación de las claves durante el procesamiento de las filas en la tarea *map*. Además de generar

las claves, la tarea *map* va a crear los pares clave-valor de la salida en los que la clave es la clave de la fila que se ha generado y el valor va a ser el objeto Put utilizado en el método de carga de HBase Put. Es decir, se va a instanciar un objeto Put por fila con la clave generada y se van a añadir cada uno de los valores de los campos de la filas a las columnas correspondientes.

La figura 3.4 muestra una representación gráfica de cada una de las fases que tienen lugar durante el proceso de carga. Para comenzar (Figura 3.4 fase 1), los datos a cargar tienen que estar almacenados en uno o más ficheros, en los cuales las columnas están separadas mediante un mismo delimitador (, ; — ....). Los ficheros tienen que ser copiados en HDFS especificando un tamaño de bloque, si se desea un tamaño de bloque diferente a 128 MB. Internamente cuando un fichero se copia en HDFS se divide en bloques de un tamaño fijo, al copiar el fichero en HDFS se crean N bloques. El número de bloques en los que se divide el fichero determina el número de tareas *map* a ejecutar durante el proceso de carga.

Una vez ha terminado este proceso, se puede comenzar con el proceso de carga. Primero se lanzan tantas tareas *map* como bloques tiene el fichero o ficheros, siguiendo con el ejemplo se lanzarán 8 tareas *map* (Figura 3.4 fase 3). Según van terminando, se mezclan las salidas de todas las tareas *map* mediante un proceso interno de MapReducepara que cada tarea *reduce* tenga el conjunto de claves con el que trabajar (Figura 3.4 fase 4). En ese momento ya puede comenzar a ejecutar las tareas *reduce* (Figura 3.4 fase 5), se van a lanzar tantas tareas *reduce* como regiones tenga la tabla que deseamos cargar. Si no se han creado regiones tras crear la tabla el número de regiones es uno y por ello solo se va a ejecutar una tarea *reduce*. Cada tarea *reduce* va a ser la encargada de generar los ficheros HFile con los datos de una región. Una vez el proceso MapReduce ha terminado, los ficheros HFiles generados se encuentran en una carpeta en HDFS creada por el proceso MapReduce para almacenar la salida de su ejecución. HBase almacena los datos de sus regiones en una carpeta exclusiva dentro del sistema de ficheros HDFS. Para que HBase pueda acceder a los HFiles generados, éstos se mueven a la carpeta de su región correspondiente (Figura 3.4 fase 6).

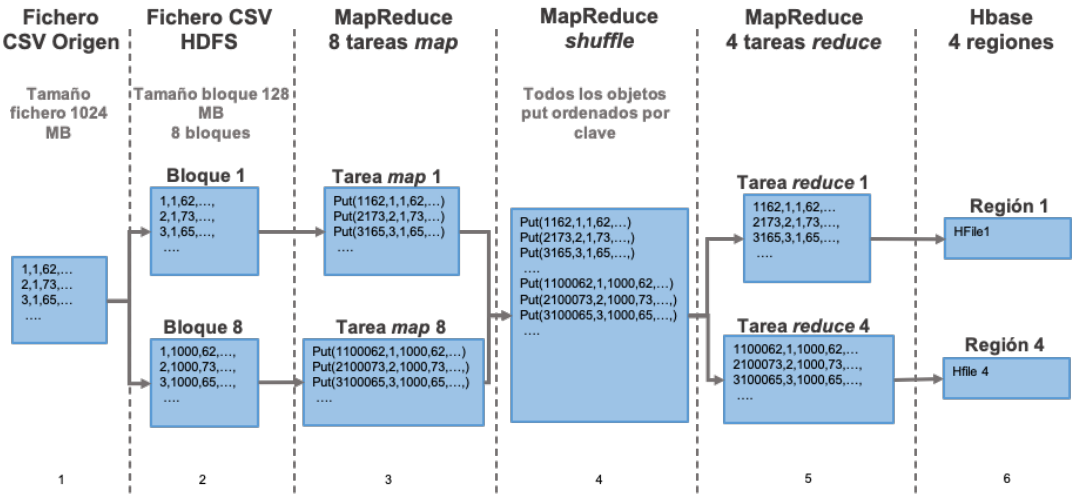


Figura 3.4: Ejemplo inyector de carga

3.4.1. Algoritmo

El algoritmo 1 describe el pseudo-código de la tarea *map*. Por cada fila que se lee del fichero se genera la clave (algoritmo 1, líneas 1 y 2). Para generar la clave se concatenan los distintos campos que la forman previamente convertidos en *arrays de bytes* teniendo en cuenta el tipo de los valores almacenados en las columnas. Por ejemplo, si el valor de una columna es un entero, el *array de bytes* generado tendrá una longitud de cuatro bytes mientras que si se trata de un *long* será de ocho, esto permite ordenar las claves de manera lexicográfica tal y como hace HBase. Si la tabla no tiene una clave definida, como es obligatorio asignarle una clave única en HBase para ello se crea una clave de 16 bytes mediante la concatenación de dos valores de tipo *long*. Estos valores de tipo *long* van desde Long.MIN_VALUE a Long.MAX_VALUE, cada vez que una nueva clave tiene que ser generada el segundo de los *long* es incrementado en uno, hasta que llega al valor Long.MAX_VALUE, momento en el que este pasa a tener el valor Long.MIN_VALUE y el primero de los *long* es incrementado en uno. Así la primera vez que se crea una clave, ésta será un *array* de 16 bytes de tal manera que los primeros ocho bytes serán los correspondientes al valor Long.MIN_VALUE y los otros ocho del *array de bytes* que represente el valor Long.MIN_VALUE. A continuación, se

crea el objeto de HBase *Put* con su correspondiente clave y los valores de las columnas (algoritmo. 1, líneas 3 a 7).

Algoritmo 1 Map

Require: columnas: el conjunto de columnas a cargar, indicado la familia de columnas y el nombre de la columna
columnasClave: columnas que forman parte de la clave
fichero: fichero a procesar
separador: caracter que separa los campos de la fila a procesar

```
1: clave ← getClaveDeLínea(fichero, separador, columnasClave)
2: claveCodificada ← generarClaveCodificada(clave)
3: put ← crearPut(claveCodificada)
4: for cada valorColumna en líneaFichero do
5:     valorColumnaCodificado ← getValorCodificado(valorColumna)
6:     addValorCodificadoAPut(put, valorCodificado, nombreColumna)
7: end for
```

La tarea *reduce* procesa cada uno de los pares clave-valor generados en la tarea *map* ordenados en orden lexicográfico por la clave y escribe los objetos *Put* en los ficheros HFile de la región correspondiente. Una vez terminado este proceso, se mueven los ficheros HFile a la carpeta de la tabla creada por HBase al crear la tabla para que HBase comience a servir los datos.

Como el inyector de carga está basado en la herramienta ImportTSV de HBase, los parámetros del inyector son similares a los de dicha herramienta: 1) el nombre de la tabla a cargar, 2) el path al fichero de datos en HDFS, 3) el nombre de las columnas como una lista de pares ”familia-de-columnas:columna”, 4) la o las columnas que forman parte de la clave en el mismo orden en el que se van a concatenar los valores y 5) el separador de campos en el fichero. El parámetro que indica las columnas que forman parte de la clave puede ser vacío en caso de que no se considere adecuadas ninguna de las columnas, en este caso se generará una clave de manera automática.

3.5. Paralelización del proceso de carga

HBase distribuye los datos entre todos sus nodos que forman parte del clúster para realizar consultas en paralelo y así proporcionar baja latencia cuando las consultas se realizan sobre un conjunto de datos muy amplio. Todos los HFiles de la misma región se almacenan en la misma carpeta y el mismo nodo del sistema de ficheros HDFS. Esta estructura interna de los datos permite paralelizar el proceso de carga siempre que la tabla haya sido dividida en regiones antes de comenzar el proceso de carga.

Como el número de regiones define el número de tareas *reduce* a ejecutar, si solo hay una región, solo se va a ejecutar una tarea *reduce*. Si la cantidad de datos a cargar es muy grande, la tarea *reduce* va a tardar mucho tiempo en finalizar y además puede ocurrir que el espacio en disco del nodo en el que se encuentre la región no sea suficiente para almacenar los datos, llegando a cancelarse el proceso de carga tal. En la parte A) de la figura 3.5 se muestra el proceso *reduce* que tiene que procesar 750 GB de datos y el espacio en disco disponible es de 500 GB, por lo que el proceso de carga se cancela tras llevar 2/3 de la ejecución.

Si dicha región se divide en dos regiones, que vayan a servir aproximadamente la misma cantidad de datos en dos máquinas. El proceso reduce se paraleliza con dos tareas reduciendo el tiempo de ejecución En la parte B) de la figura 3.5 se muestra la misma cantidad de datos a cargar empleando dos procesos *reduce* que son ejecutados en paralelo de tal manera que cada uno procesa 375 GB de datos.

3.5.1. Creación de regiones en HBase con el método *Split*

Para crear regiones de manera manual, HBase proporciona un método llamado *split*. Este método divide la tabla o región por la clave que se indique. Por ejemplo, si las claves de los datos a cargar comienzan por una letra que va desde la “A” hasta la “Z”, seguido de un número entero que va desde 0 hasta 1000. En total, se cargarán 27000 filas (27 letras en el alfabeto x 1000 tuplas cada una) de las cuales 1000 claves comienzan por la letra “A”, otras 1000 por la letra “B” y así sucesivamente, tal y como se muestra en la figura 3.6. Para crear dos regiones del mismo tamaño, hay que utilizar

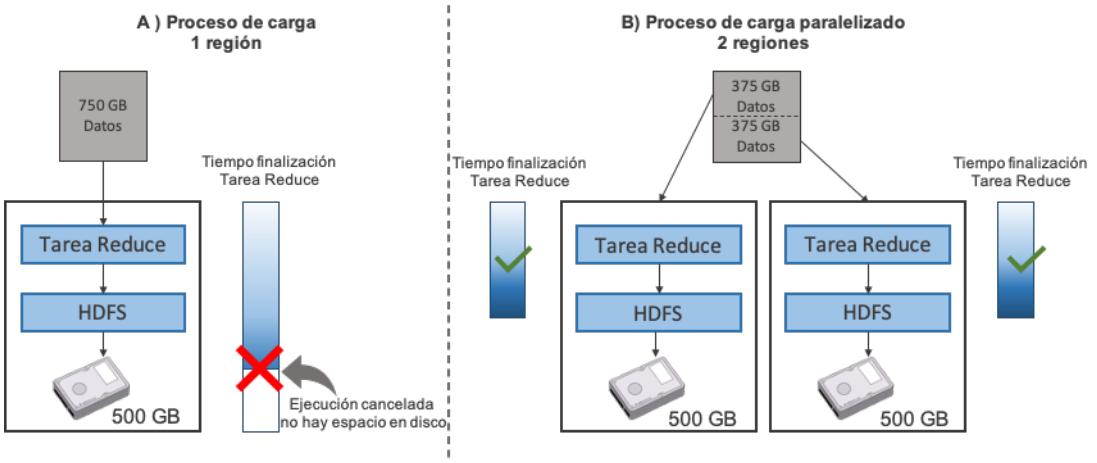


Figura 3.5: Comparación carga de datos en una región y en dos regiones

el método *split* indicando el nombre de la tabla o el identificador de la región a dividir y la clave. En la figura 3.6, se muestra la Región 1 vacía ya que el proceso de carga no ha comenzado y ésta recibe el comando "Split 'region1','M0500"' que divide la Región 1 en dos nuevas regiones Región 1-1 y Región 1-2 las cuales se encuentran vacías y cuyas claves de inicio y fin son "[,M0500)" y "[M0500,)". De esta manera cuando se cargue la tabla las filas desde la clave "A0001" a "M0499" irán a la Región 1-1 y desde la clave "M0500" a "Z1000" irán a la Región 1-2.

Para hacer esta distribución de los datos en regiones hay que conocer la distribución de las claves a cargar. Cuando los datos a cargar son poco, se pueden analizar previamente a la carga. Pero cuando hay cargar grandes cantidades de datos y no es factible el análisis, se hace muy complicado crear regiones con un mismo número de claves aproximadamente.

Para crear regiones antes de realizar la carga de una tabla sin tener conocimiento de la distribución de las claves, HBase tiene un clase Java llamada *RegionSplitter* que utiliza dos algoritmos, *DecimalStringSplit*, *HexStringSplit*. El primer algoritmo considera claves de dígitos codificadas como *long* en el rango "00000000" => "99999999" o alguna otra distribución uniforme, y estos valores son rellenados con ceros a la izquierda para que todas las claves tengan la misma longitud y tengan el mismo orden lexicográfico. Es decir, la clave con valor 1 es "00000001" y la clave con valor 10 es "00000010"

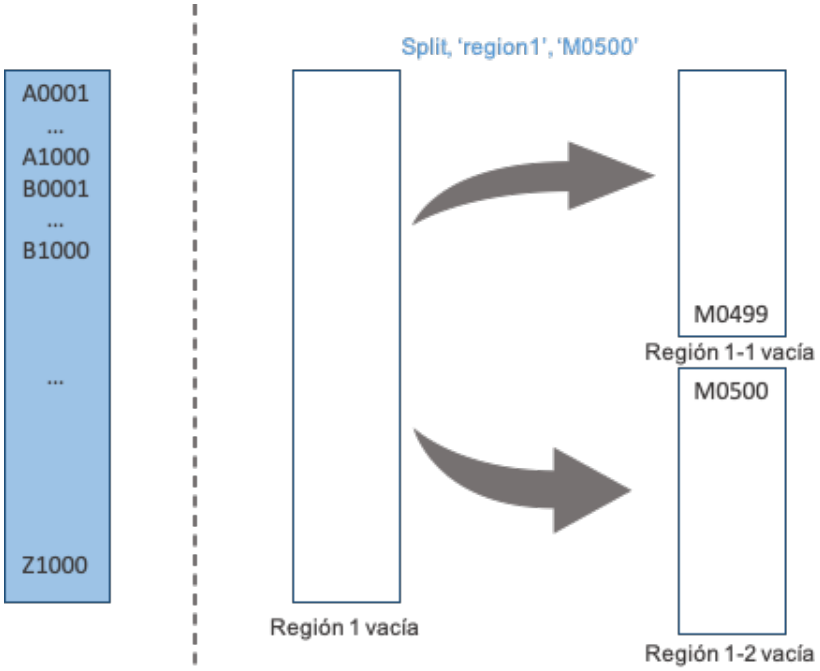


Figura 3.6: Creación de regiones en HBase método Split

de esta manera se asegura que la clave 10 después de la clave 9 ,“00000009”, y no de la clave 1. Por lo que el algoritmo *DecimalStringSplit* creará dos regiones de tal manera que la primera irá desde la clave “00000000” hasta “49999999” y la segunda región desde “50000000” hasta “99999999”. El segundo algoritmo, *HexStringSplit*, sigue el mismo comportamiento que el anterior pero en este caso las claves son una representación en ASCII de las claves codificadas con el algoritmo de encriptación MD5 o cualquiera otra distribución uniforme con valor hexadecimal en el rango “00000000” => “FFFFFFFF”. Por ejemplo, si se desea crear dos regiones de la tabla utilizando el algoritmo *HexStringSplit*, la primera región tendrá las claves en el rango “00000000” => “77777776” y la segunda región en el rango “77777777” => “FFFFFFFF”.

3.5.2. Creación de regiones equilibradas

Ambos algoritmos no aseguran que se vayan a crear regiones con un número de claves aproximadamente igual, en ambos casos se asume que la distribución de las claves tiene que ser uniforme. Pero en caso de no ser uniforme puede ocurrir que se creen

regiones con 1 fila y otras que tengan millones de filas. Por ello se ha implementado una herramienta que permite crear regiones, las cuales tendrán aproximadamente la misma cantidad de filas tras finalizar la carga. Esta herramienta requiere acceder a la descripción de la tabla y conocer cómo es la clave en caso de tenerla, es decir, si no se especifica que campo o campos forman parte de la clave, esta se generará de manera automática. La cantidad de regiones a crear viene definida por el número de HBase region servers para asegurar el máximo paralelismo posible durante la carga. De esta manera se distribuye el trabajo del proceso de carga entre todas las máquinas, procesando cada una de ellas aproximadamente la misma cantidad de datos.

Para dividir las claves en regiones con el mismo número de claves habría que recorrer el fichero que se va a cargar para conocer todo el espacio de claves y así poder dividirlo en regiones que tengan el mismo número de filas y a continuación volver a recorrer el fichero para realizar la carga de los datos. Cuando el fichero es de tamaño de cientos de gigabytes o incluso de terabytes este proceso no es factible por el tiempo necesario para recorrer dos veces los datos. Para evitar esto se va a obtener un conjunto de muestras del fichero del cual se va a obtener las claves por las que se va a crear las regioens. Así se evita tener que recorrer dos veces el fichero.

Para saber cuál es el número de muestras que hay que obtener del fichero se ha utilizado la formula 3.1 que permite obtener el tamaño de la muestra desconociendo el tamaño de la población, es decir sin saber el número de filas del fichero (Teorema Cochran [39]).

$$\#Muestras = \frac{Z^2 * p * q}{d^2} \tag{3.1}$$

Siendo Z_α = nivel de confianza, p = probabilidad de éxito, q = 1 - p = probabilidad de fracaso y d = precisión, por ejemplo 1 %. Supongamos que deseamos conocer el número de muestras que hay que obtener del fichero si no sabemos el tamaño del mismo, para ello fijamos los valores Z^2 = 1.962 (ya que la seguridad es del 95 %), p =

50 % = 0,5, q = 1 – p = 1 – 0.5 = 0,5, d = precisión = 3 %. Reemplazando estos valores en la fórmula 3.1 se obtiene que el número de muestras a obtener es de 1086.

El número de muestras es el primer paso del algoritmo para crear regiones equilibradas, para ello hay que indicar el nivel de confianza, la probabilidad de éxito y la precisión, el path al fichero que se va cargar y la composición de clave, es decir una cadena de nombres de campos separado por coma. Utilizando la fórmula 3.1, se obtiene el número de muestras del fichero (n) (algoritmo 2, línea 1). Estas muestras se obtienen de n puntos equidistantes del fichero, para ello se divide el tamaño del fichero entre el total de muestras a obtener, lo cual indica cada cuantos bytes del fichero hay que obtener la muestra. Como puede ser que la posición a ser leída del fichero caiga en mitad de una línea, se obtiene la línea siguiente (algoritmo 2, línea 2). Para cada fila o muestra, se genera la clave leyendo los valores de las columnas que forman parte de la claves Las claves son ordenadas siguiendo el orden lexicográfico, obteniendo una lista de claves ordenada (algoritmo 2, línea 3). Una vez ha terminado este proceso, se seleccionan tantas claves como número de regiones a crear menos uno, estas son obtenidas siguiendo un muestreo sistemático definido por *#Muestras / #Regiones* (línea 4, algoritmo 2). Con las claves obtenidas las regiones son creadas utilizando el método *split* de HBase y son distribuidas entre los HBase region servers siguiendo una distribución en “round-robin” entre todos los HBase region server (líneas 6, algoritmo 2). Para mover las regiones desde un HBase region server a otro, se utiliza el método *move* de HBase al cual hay que indicar el identificador de la región que se desea mover y el HBase region server de destino.

Este algoritmo permite crear regiones de tal manera que cada una de ellas tenga aproximadamente la misma cantidad de filas.

Algoritmo 2 Pre-Split

Require: nivelConfianza: Nivel de confianza para calcular la muestra
Require: probExito: Probabilidad de éxito para calcular la muestra
Require: precisión: Precisión para calcular la muestra
Require: fichero: Path al fichero que se va a cargar
Require: composicionClave: los campos de forman parte de la clave

```
1: #Muestras ← get#Muestras(nivelConfianza, probExito, precisión)
2: muestras ← getMuestrasFicheroCSV(fichero, #Muestras)
3: clavesOrdenadas ← generarYOrdenarClaves (muestras, composiciónClave)
4: claves ← getClaves (calvesOrdenadas)
5: splitTabla(claves)
6: distribuirRegiones()
```

3.6. Distribución de recursos de las máquinas del sistema

La distribución de recursos permite aprovechar al máximo las características de cada una de las máquinas y de esta manera obtener el máximo paralelismo posible durante el proceso de carga. Otra de las contribuciones de este capítulo es el reparto de los recursos de cualquier máquina que tenga arquitectura NUMA entre todos los servicios necesarios para realizar la carga de datos en HBase. Los servicios necesario para usar HBase son ZooKeeper (ZK), HDFS (HDFS NameNode y HDFS DataNode) y HBase (HBase HMaster y HBase region server) y los servicios necesarios para realizar la carga son YARN (YARN ResourceManager y YARN NodeManager) que permiten gestionar los recursos asignados a MapReduce todos ellos presentados en e capítulo 2. Todos estos servicios se despliegan en un cluster con varios nodos tal y como se muestra en la parte a) de la figura 3.7. Aunque estos servicios están pensados para ser distribuidos, hay máquinas que tienen una arquitectura NUMA congran cantidad de recursos tanto en RAM como en cores en la cual se pueden desplegar todos los servicios, aislándolos en distintos nodos NUMA, cómo se muestra en la parte b) de la figura 3.7. En el caso del clúster de nodos uno de ellos va a ser el encargado de ejecutar todos los procesos de control de los servicios (ZK, HDFS NameNode, YARN ResourceManager y HBase Master), a este nodo se le ha llamado “Nodo de control”.

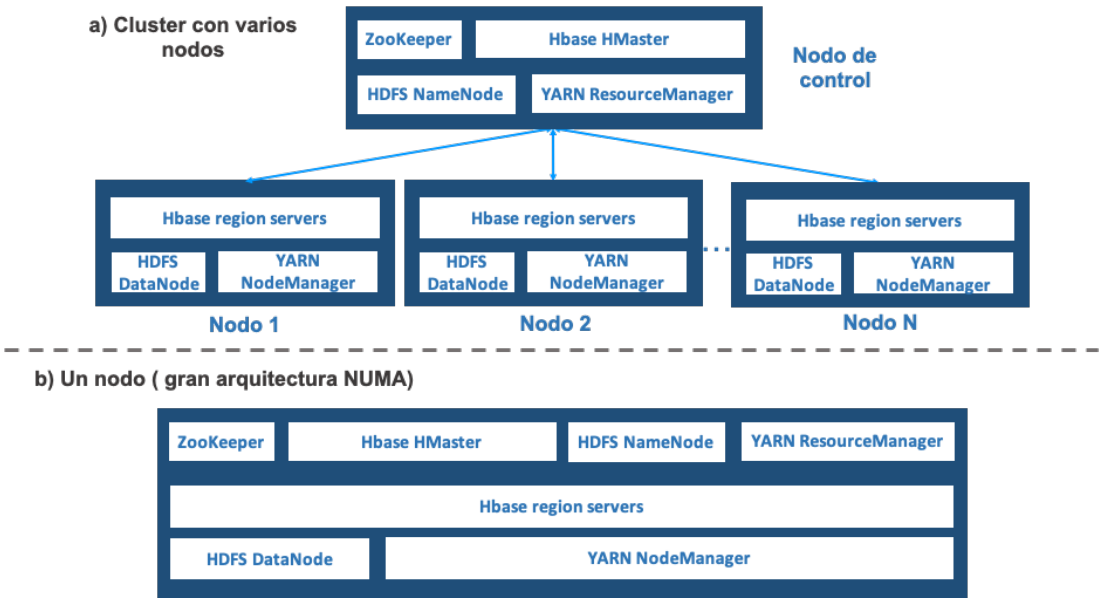


Figura 3.7: Distribución de los servicios para el inyector de carga.

Para el entorno de la máquina con una gran arquitectura NUMA se van a aislar todos estos servicios colocando su ejecución en uno de los nodos NUMA y el resto de servicios (HDFS DataNode, HBase region server y YARN NodeManager) se van a colocar en el resto de nodos NUMA de la misma, tal y como se muestra a continuación. En cuanto al despliegue del clúster el resto de servicios se ejecutaran de manera replicada en cada uno de los nodos (“Nodo de datos”).

La distribución de recursos es una tarea compleja cuando la arquitectura NUMA de la máquina a configurar es muy grande, con muchos recursos, por ejemplo 4 terabytes de memoria RAM y 512 cores virtuales. La asignación de recursos (memoria RAM y cores virtuales) que viene predefinidos en los distintos servicios no son siempre los más adecuados para las diferentes máquinas que podemos encontrar en el mercado, haciendo que no se aprovechen todos los recursos que nos ofrecen.

Las máquinas que se vayan a configurar deben de cumplir dos condiciones: 1) Al menos tiene que haber un disco en el nodo donde se almacenaran los datos para evitar latencias extra al acceder a los datos que se vayan a almacenar y 2) Al menos tiene que haber un core por servicio, es decir que cada servicio tiene que tener su propio core de

tal manera que no tenga que compartir recursos con ningún otro servicio. Para facilitar la tarea de distribución de recursos y asignación de los servicios a los distintos cores de las maquinas, se ha diseñado un algoritmo.

El algoritmo para la asignación de recursos se divide en tres pasos diferenciados por el servicio que se va a configurar en cada uno de ellos. Primero comienza con la distribución y asignación de cores al servicio HDFS DataNode (HDFS_DN), este servicio necesita acceder directamente a los discos, ya que es el encargado de almacenar de manera persistente los datos de las tablas de HBase. Por ello, este servicio se despliega en los cores del nodo NUMA que tenga menor latencia de acceso a los discos. En caso de que haya varios nodos NUMA con discos se desplegará en el nodo NUMA con discos más potentes, en caso de que sean todos iguales se asignará al uno de los nodos NUMA con discos. Una vez realiza esta asignación, se asigna el servicio YARN NodeManager (YARN_NM), de tal manera que se encuentre en ejecución en el core o cores que tenga una latencia de acceso al nodo NUMA en el que se encuentra el servicio HDFS DataNode lo más baja posible. Esto es debido a que el proceso YARN NodeManager va a ser el encargado de procesar el o los ficheros con los datos a cargar en HBase y va a almacenar tras la ejecución del proceso MapReduce los datos en HDFS. Finalmente, se asignarán el resto de cores y recursos disponibles al servicio HBase region server (HBase_RS). En una misma máquina puede haber más de un HBase region server, por tanto estos recursos se repartirán de manera equitativa entre todos los servicios HBase region server.

El algoritmo de distribución de recursos 3 necesita: 1) lista de los servicios a desplegar en el nodo. 2) La distribución de los cores virtuales y de las tarjetas de memoria. 3) La lista de nodos NUMA que tienen directamente conectados uno o más discos. La lista de nodos NUMA con uno o más discos conectados directamente se obtiene mediante el comando *numactl –hardware* que muestra a la arquitectura NUMA y la matriz de latencia de acceso a la memoria RAM. El comando *lshw -c storage -c disk* muestra el identificador del PCI del controlador SATA al que está conectado el disco. El identificador permite conocer el nodo NUMA al que está conectado mediante la lectura del fichero de sistema /sys/bus/pci/device/pciSATAControllerID/numa_node.

Algoritmo 3 Distribución de los recursos entre los servicio según la arquitectura NUMA del nodo

Require: ListaServicios: Lista de los servicios a desplegar
arquitecturaNUMA: La arquitectura NUMA de la máquina a configurar
ListaNodosNUMAconDisco: Lista de nodos NUMA con discos directamente conectados

```
1:  if #Nodos_NUMA_Con_Discos > 0 then
2:      if #servicios <= #Nodos_NUMA then
3:          AsignaNodoNUMAyRAMaHDFS_DN(ListaNodosNUMAconDisco)
4:          #NNaYARN_NM=#NN-1-#HBase_RS
5:          AsignaNodoNUMAyRAMaYARN_NMminDistanciaHDFS_DN(#NNaYARN_NM, arquitecturaNUMA)
6:          AsignaNodoNUMAyRAMaCadaHBase_RS(#NNaHBase_RS, arquitecturaNUMA)
7:      else if #servicios <= #Cores then
8:          #CoresPorServicio=getCoresPorServicio(#Cores, ListaServicios)
9:          AsignaCoresyRAMaHDFS_DN(ListaNodosNUMAconDisco)
10:         AsignaCoresyRAMaYARN_NMminDistanciaHDFS_DN(#CoresYARN_NM, arquitecturaNUMA)
11:         AsignaCoresyRAMaCadaHBase_RS(#CoresToHBase_RS, arquitecturaNUMA)
12:     else
13:         Error: Al menos tiene que haber un core por servicio.
14:     end if
15: else
16:     Error: Al menos tiene que haber un disco.
17: end if
```

Dependiendo de la arquitectura de la máquina que se va a configurar, se pueden dar dos escenarios: 1) el número de servicios a desplegar es menor o igual al número de nodos NUMA, por lo que tendremos a los servicios aislados en distintos nodos NUMA, asignando al servicio todos los cores que comprenden el nodo NUMA seleccionado. 2) El número de servicios es mayor que el número de nodos NUMA, en este caso habrá más de un servicio por nodo NUMA (algoritmo 3 líneas 2 y 7). En ambos escenarios el orden de distribución de los recursos será el mismo, HDFS_DN, YARN_NM y HBase_RS. En el primer escenario, en el que hay al menos tantos nodos NUMA cómo servicios (algoritmo 3 líneas 2 a 6), primero se asignan los recursos del servicio HDFS_DN para ello se escoge un nodo NUMA de la lista de nodos NUMA que tienen directamente conectados uno o más discos. En caso de tener más de un nodo NUMA con disco conectado se selecciona el que tenga los discos de mayor capacidad y mayor tasa de escritura y lectura, y se el asigna la configuración de ese nodo NUMA al servicio. Por ejemplo, en la figura 3.8 en la que se muestra la arquitectura NUMA

de una máquina con 8 nodos NUMA (64 cores virtuales) y 128 GB de RAM en la que el nodo NUMA con identificador #0 tiene directamente conectados dos discos y en la parte derecha de la figura se muestra la matriz de distancias de acceso a memoria. El servicio HDFS_DN se ejecutará en el nodo NUMA #0 que es el que tiene conectados los discos, y se le asigna toda la memoria RAM de dicho nodo NUMA ya que ese proceso va a ser el único en ejecución en dicho nodo y por lo tanto no tiene que compartir recursos con otros servicios (16 GB) y se asociará el proceso a los cores pertenecientes a dicho nodo NUMA, cores virtuales del 0 al 7 (algoritmo 3, línea 3).

A continuación se calculan los nodos NUMA que se van a asignar al proceso YARN_NM, restando del total de nodos NUMA de la arquitectura 1 por haber sido asignado al servicio HDFS_DN y tantos como servicios HBase_RS se vayan a desplegar (algoritmo 3, línea 4).Se asigna al servicio YARN_NM 3 nodos NUMA ya que los 4 nodos NUMA restantes van a ser asignados a los HBase_RS. Se comienzan a asignar los nodos para la ejcución del servicio YARN_NM por aquellos nodos NUMA que tengan la mínima distancia al nodo NUMA donde se va a ejecutar el servicio HDFS_DN, es decir los nodos NUMA #2, #4 y #6 (algoritmo 3, línea 5). Por lo tanto el servicio YARN_NM se va a configurar con los siguientes recursos: 24 cores virtuales(16-23, 32-39 y 48-55) y 48 GB de RAM, la cantidad de RAM directamente conectada a los nodos NUMA asignados. Para finalizar se asignan el resto de nodos NUMA y la memoria RAM correspondiente a cada uno de los HBase_RS que van a ejecutarse en la máquina (algoritmo 3 línea 6). Tras distribuir los nodos NUMA de la máquina de la figura 3.8, los 4 servicios HBase region server se asignarán a los nodos NUMA #1 (cores 8-15), #3 (cores 24-31), #5 (cores 40-47) y #7 (cores 53-63) respectivamente, con 16 GB de RAM cada uno.

En el segundo escenario, en el que no hay suficientes nodos NUMA para todos los servicios (algoritmo 3 líneas 7 a 11), primero se calcula el número de cores que van a ser asignados a cada uno de los servicios (algoritmo 3, línea 8), para ello se necesita conocer la arquitectura NUMA ya que el servicio HDFS_DN va a ser colocado en el nodo NUMA que se tenga directamente conectado algún disco y también sabemos que el servicio YARN_NM tiene que ejecutarse en un nodo NUMA cuya latencia al

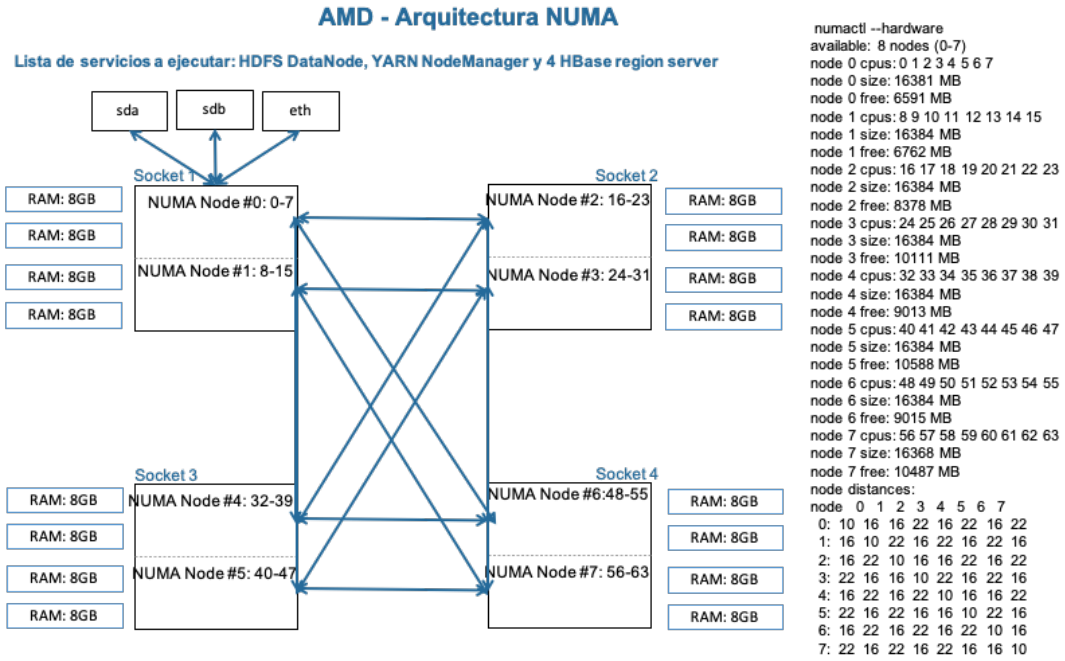


Figura 3.8: Distribución de los servicios para el escenario 1

acceso de memoria donde se encuentra el servicio HDFS_DN sea la mínima posible. Por ejemplo, en la figura 3.9, se muestra una máquina con 2 nodos NUMA (24 cores virtuales), en el que uno de ellos tiene conectado directamente tres discos y en la parte de la derecha se muestra la matriz de distancias de acceso a memora. En este caso el nodo NUMA que tiene directamente conectados los discos será el que vayan a compartir los servicios HDFS_DN y YARN_NM. Como el mayor procesamiento se va a realizar en el servicio YARN_NM, se va a signar al servicio HDFS_DN un core físico con su correspendiente cantidad de memoria y el resto de cores de ese nodo NUMA serán asignados junto con la memoria al servicio YARN_NM. Es decir, siguiendo con la figura 3.9, el servicio HDFS_DN se va a configurar con 11GB de RAM y se va a asignar a 2 cores virtuales (cores 0 y 12) y el resto, 10 cores virtuales y 53 GB de RAM se va a asignar al servicio YARN_NM.

Los cores del otro nodo NUMA serán divididos entre todos los servicios HBase_RS que tiene que desplegarse. Es decir, si se van a desplegar 4 HBase_RS y tenemos 12 cores virtuales y 64 GB de RAM en el nodo NUMA #1, se va a asignar a cada servicio

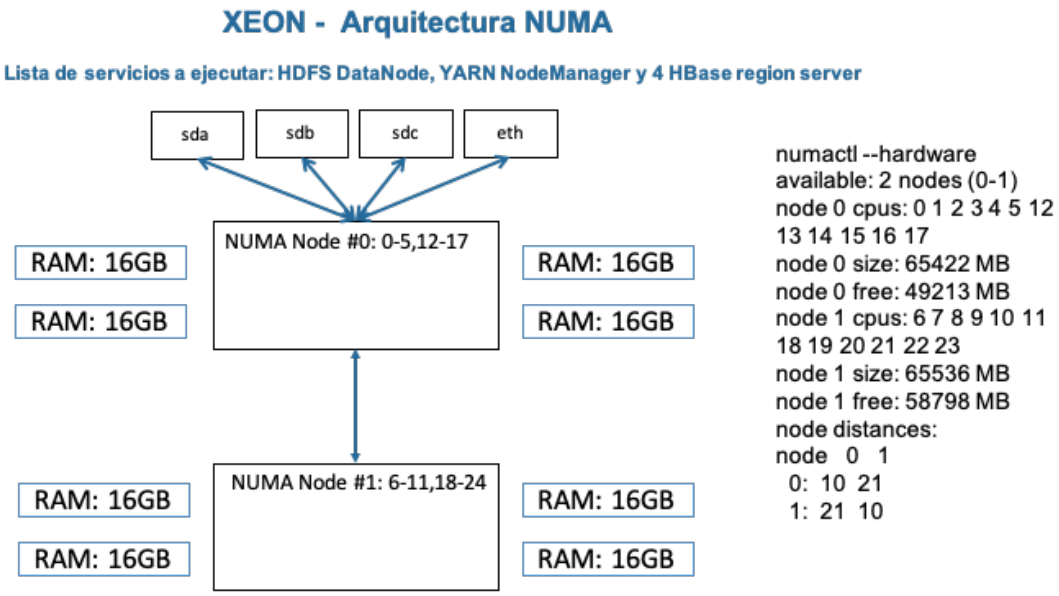


Figura 3.9: Distribución de los servicios para el escenario 2

HBase_RS 2 cores virtuales y 16 GB de RAM a cada uno. De esta manera el primer HBase_RS será asignado a los cores (6 y 18), el segundo a los cores (7 y 19) y así sucesivamente (algoritmo 3 líneas 9 a 11).

Con este algoritmo se puede distribuir los recursos de cualquier máquina con arquitectura NUMA entre una lista de servicios definida permitiendo aprovechar al máximo las capacidades (Memoria, CPU y acceso a disco) de las máquinas de un clúster. En este caso se ha utilizado el algoritmo para distribuir los recursos de las máquinas que forman parte de un clúster, las cuales van a ejecutar los servicios HDFS DataNode, HBase region server y YARN NodeManager, pero del mismo modo puede ser utilizado para distribuir los recursos de otros servicios.

3.7. Distribución de los recursos del servicio YARN NodeManager

YARN, tal y como se ha descrito en el capítulo 2, es un gestor de recursos para MapReduce que permite configurar los recursos de memoria y cores que van a ser

utilizados por cada uno de los procesos y de las tareas. El servicio YARN ResourceManager es el encargado de la negociación de los recursos necesarios para lanzar el primer contenedor el cual va a ejecutar el *Application Master*. Un contenedor es un conjunto de recursos (RAM y cores virtuales) reservados para ejecutar uno de los tres servicios diferentes que están disponibles en YARN: *Application Master*, *Map* y *Reduce*. El *Application Master* se encarga de la negociación en la creación de contenedores según los recursos disponibles en cada una de las máquinas, teniendo en cuenta los parámetros de configuración especificados en los ficheros yarn-site.xml y mapred-site.xml. Cuando una tarea *map* o *reduce* tiene que ser ejecutada, el *Application Master* define un contenedor con la configuración especificada en el fichero mapred-site.xml, los contenedores de cada uno de los dos tipos de tareas tiene asignada una cantidad de RAM y CPU.De esta manera los recursos son exclusivamente utilizados por dicha tarea y cuando la tarea termina los recursos son liberados para poder ser utilizados en la creación de otro contenedor. La configuración de los contenedores puede ser diferente para cada uno de los procesos: *Application Master*, *Map* y *Reduce*. A lo largo de esta sección se va a mostrar la configuración predefinida de YARN, la configuración obtenida mediante la herramienta *yarn-util.py*[38] y una de las aportaciones de este capítulo, la configuración de YARN que permite obtener el máximo rendimiento durante el proceso de inyección de carga.

Una configuración no adecuada de los recursos de los contenedores *map* puede hacer que el tiempo total del proceso de carga se vea incrementado debido a la falta de contenedores disponibles para procesar todos los bloques en los que ha sido dividido el fichero. La figura 3.10 muestra una representación en el tiempo de la ejecución del trabajo *MapReduce*, para realizar la carga de un fichero que ha sido divido en 10 bloques en una tabla con 4 regiones.En la parte superior se muestran los contenedores *map* agrupados en iteraciones las cuales muestran el total de contenedores que puede ejecutarse en paralelo, 4 en este caso. Por ello primero se procesarán 4 bloques, en una segunda iteración se procesarán otros 4 bloques y finalmente se procesarán los dos restantes. Durante la ejecución de la tercera iteración de tareas *map*, hay 40960 MB que se encuentran disponibles. En ese momento el proceso *Application Master* detecta

que hay procesos *reduce* a ejecutar y que hay suficientes recursos para crear dos contenedores *reduce*, parte inferior de la figura 3.10. Hasta que el procesamiento de todos los bloques no haya terminado, las tareas *reduce* no pueden finalizar su ejecución debido a que necesitan procesar todos los datos intermedios generados por las tareas *map*, de ahí que la ejecución de los primeros contenedores *reduce* sea más prolongada en comparación con la ejecución de los contenedores *reduce* 3 y 4 que comienzan inmediatamente después de finalizar los contenedores *map* al haberse quedados disponibles suficientes recursos para su ejecución.

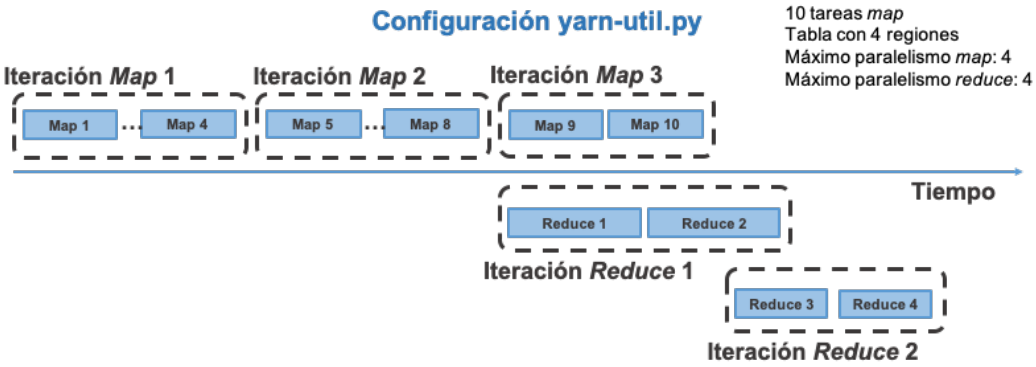


Figura 3.10: Representación de ejecución de las tareas *map* y *reduce* en el tiempo

3.7.0.1. Configuración de YARN para el Inyector de Carga

Para encontrar la configuración más apropiada para YARN teniendo en cuenta los recursos asignados al servicio YARN NodeManager se ha diseñado un algoritmo que además tiene en cuenta el tamaño de los bloques de HDFS y el número de regiones de las tablas para conseguir un mayor paralelismo y por ende un mejor rendimiento del proceso de carga al distribuir la ejecución equitativamente entre todos los nodos del sistema.

El número de tareas *map* total que tienen que ser ejecutadas viene definido por el número de bloques del fichero en HDFS y el número de bloques depende directamente del tamaño especificado al cargar el fichero a HDFS. Debido a la importancia que

tiene la correcta configuración de los recursos asignados para cada uno de los contenedores *map* para conseguir el mayor paralelismo posible durante la ejecución del proceso de carga, se ha realizado un estudio para intentar definir cual es el tamaño de bloque más adecuando teniendo en cuenta el fichero que va a ser cargado y el tipo de almacenamiento. Lo mismo ocurre con los contenedores de las tareas *reduce*, cuyo número es definido por el número de regiones de la tabla y de ahí que la distribución de los recursos para estos contenedores se tiene que realizar de manera que se puedan ejecutar tantos contenedores *reduce* en paralelo como regiones haya en cada máquina del clúster para poder alcanzar el mayor paralelismo y así cargar los datos en el menor tiempo posible.

El impacto del tamaño de los bloques en HDFS

El mínimo tamaño de bloque en HDFS es 32 MB y por omisión el tamaño de bloque es de 128 MB. Estos valores pueden generar muchos bloques si el tamaño del fichero es muy grande, lo que puede aumentar el tiempo de ejecución de la carga concretamente de la fase *map* debido a que hay tantas tareas *map* como bloques y al tiempo que se tarda en inicializar cada una de ellas, de ahí que el tamaño de los bloques sea tan importante. Por ejemplo, con un fichero de 10 GB y un tamaño de bloque de 32 MB, se crean 320 bloques. Si el clúster utilizado durante el proceso de carga no puede ser configurado con un paralelismo de 320 contenedores, el número de iteraciones, veces que se crean contenedores para ejecutar tarea, que van se van a ejecutar es mayor que una, además del tiempo necesario en YARN para crear cada uno de los contenedores *map*.

Además del impacto del tamaño de bloque en el número de tareas *map* también tiene impacto en el tiempo en cargar el fichero en HDFS. El tiempo que se tarda en almacenar el fichero en HDFS varía dependiendo del tamaño bloque debido ancho de banda, la tasa de escritura en disco y red mediante la que están conectadas las máquinas del clúster. Se ha evaluado el tiempo de carga en HDFS dependiendo del tamaño de bloque, del dispositivo en el que se almacenan los datos y la conexión entre

las máquinas (ethernet a 1 Gb e infiniband a 10 Gb). Esta evaluación ha sido realizada utilizando un disco magnético de tipo HDD y dos discos con diferentes características de tipo SSD. El disco HDD tiene una tasa de transferencia máxima de 175 MB/s, esto significa que tanto las lecturas como las escrituras se hacen a una tasa de 175 MB/s si el fichero no está fragmentado. En todas las máquinas utilizadas para estas evaluaciones los discos han sido configurados con ext4 [40], eso significa que mientras haya suficiente espacio en disco el fichero no va a ser fragmentado. En cuanto a los discos SSD, el primer modelo (SSD 1) es Intel 3500 480 GB con una tasa de lectura 500 MB/s y una tasa de escritura de 410 MB/s. El segundo modelo (SSD 2) es Intel 3510 480 GB con una tasa de lectura de 500 MB/s y una tasa de escritura de 440 MB/s.

Para comparar los distintos discos, los servicios HDFS DataNode han sido configurados para guardar los datos en uno de los discos SSD y los datos iniciales están en el disco HDD o en otro disco SSD para comparar el impacto que tiene la localización inicial del fichero durante el proceso de carga. El fichero original se encuentra almacenado en un nodo que no tiene ningún servicio HDFS DataNode ejecutándose en él, durante el proceso de carga los datos de ese fichero son almacenados entre los distintos nodos que tiene el servicio HDFS DataNode ejecutando, tal y como se muestra en la figura 3.11. Los bloques en los que es dividido el fichero se van almacenando en las distintas máquinas de tal manera que cada una de ellas va a tener aproximadamente el mismo número de bloques. Las figuras 3.12 y 3.13, muestran el tiempo de carga de dos ficheros de 28 GB (28672 MB) y 169 GB (173056 MB), respectivamente, a HDFS usando tamaños de bloque desde 1024 MB a 3027 MB.

La figura 3.12 muestra el tiempo de carga del fichero desde el disco HDD y la figura 3.13, muestra el mismo resultado cuando los ficheros están en el otro disco SSD estando todas las máquinas conectadas mediante una red ethernet a 1 Gb. Observando la gráfica se puede ver que el tiempo de carga a HDFS permanece estable hasta un tamaño de bloque de 2048 MB. En el caso del fichero de 28 GB el tiempo que ha tardado aproximadamente 275 segundos, 1570 segundos para el fichero de 169 GB independientemente de las características de disco SSD en el que se almacenan los bloques del fichero. A partir de los 2048 MB de tamaño de bloque, el tiempo de carga se

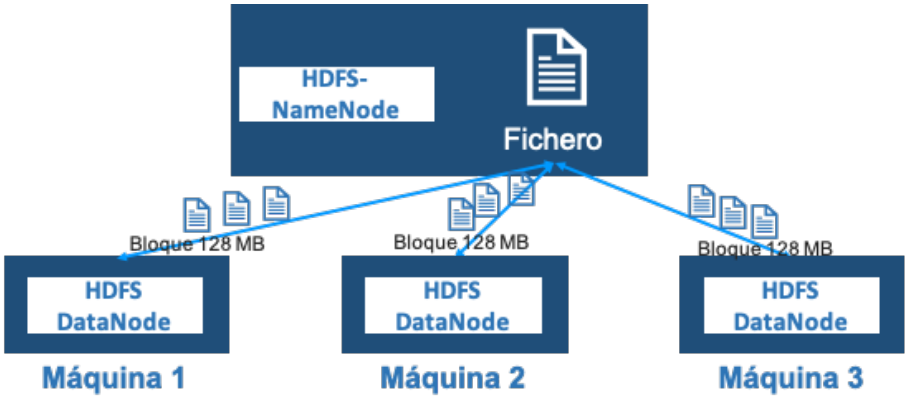


Figura 3.11: Representación fichero almacenado en un clúster de tres máquinas

ve incrementado notablemente y, además, se observa una gran diferencia dependiendo del tipo de disco. Con el fichero de 28 GB, el disco SSD con una tasa de escritura de 410 MB/s (SSD1 en la leyenda) tarda 950 y 1468 segundos para los tamaños de bloque de 2560 y 3072 MB, respectivamente. Para el fichero de 169 GB estos tiempos se ven incrementados, llegando a tardar 4810 y 6780 segundos con los mismos tamaños de bloque. Comparando estos tiempos con los obtenidos con los discos SSD con una tasa de escritura de 440 MB/s (marcador cuadrado, SSD2 en la leyenda), estos descienden en aproximadamente un tercio en todos los casos. Para el fichero de 28 GB los tiempos obtenidos son de 579 y 824 segundos y para el fichero de 169 GB los tiempo son de 3473 y 4703 segundos para los tamaños de bloque de 2560 y 3072 MB. Por lo tanto, la tasa de escritura en disco repercute en el tiempo de carga del fichero a HDFS, siendo notable la diferencia a partir de un tamaño de bloque de 2048 MB.

Por otro lado en la figura 3.13 se muestra la gráfica con los tiempos de carga de esos mismos dos ficheros desde un disco SSD a discos SSD. En este caso se ha comparado el ancho de banda de la red comparando la conexión entre máquinas mediante una red ethernet a 1Gb e Infiniband (Inf. en la leyenda) a 10Gb (líneas a rayas). Tal y como ocurre en la figura 3.12, el tiempo de carga del fichero a HDFS con un tamaño de bloque igual o inferior a 2048 MB apenas varía utilizando distintos tamaños de bloque. En el caso del fichero de 28 GB se tarda unos 276 segundos cuando las máquinas se encuentran conectadas mediante ethernet a 1 Gb y 1600 segundos para el fichero de

169 GB, independientemente de la tasa de escritura del disco. A partir de dicho tamaño de bloque el tiempo aumenta y se aprecia la diferencia entre las tasas de escritura de los discos SSD. Para el disco SSD con una tasa de escritura de 410 MB/s (SSD1 en la leyenda) para los tamaños de bloque de 2560 y 3072 MB el tiempo que tarda en cargar el fichero es de 848 y 1205 segundos para el fichero de 28 GB y de 4904 y 6940 segundos para el fichero de 169 GB. Comparando estos tiempos con los resultados obtenidos con el disco SSD con una tasa de escritura de 440 MB/s (SSD2 en la leyenda), el tiempo para cargar los ficheros en HDFS disminuye un tercio, 585 y 837 segundos para el fichero de 28 GB y de 3532 y 4813 segundos para el fichero de 169 GB.

Por otra parte si comparamos los tiempos obtenidos utilizando la red Infiniband a 10 Gb como conexión entre las máquinas y el mismo disco SSD con una tasa de escritura de 440 MB/s, éstos son tres veces menores a los obtenidos con la red ethernet a 1Gb. Al cargar el fichero de 28 GB se han obtenido los siguientes tiempos: aproximadamente 90 segundos para tamaños de bloque menor o igual a 2048 MB, 435 segundos con tamaño de bloque de 2560 MB y 637 segundos indicando 3072 MB como tamaño de bloque. Para el fichero de 169 GB los tiempos son 1568 segundos para tamaños de bloque menores o igual a 2048 MB y 3473 y 4703 segundos para 2560 y 3072 MB, respectivamente.

Comparando ambas figuras, 3.12 y 3.13 en los resultados obtenidos utilizando ethernet a 1 Gb como conexión entre las máquinas, no se aprecia una gran diferencia entre leer el fichero a cargar en HDFS desde el disco magnético y desde el disco SSD. Para el fichero de 28 GB con tamaños de bloque menores a 2560 MB, no hay diferencia. Para el fichero de 169 GB el tiempo obtenido es 1568 segundos de media en ambos escenarios.

Por lo tanto analizando los datos obtenidos de esta evaluación, el principal cuello de botella al cargar los ficheros a HDFS es la tasa de escritura el disco en el se almacenan los datos en primera instancia y la red que conecta todas las máquinas del clúster en segundo lugar.

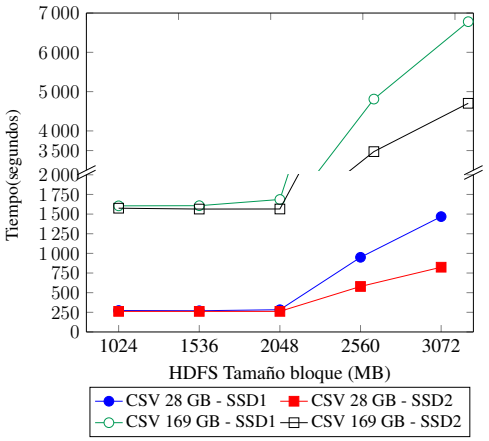


Figura 3.12: Tiempo de carga en HDFS de HDD a SSD

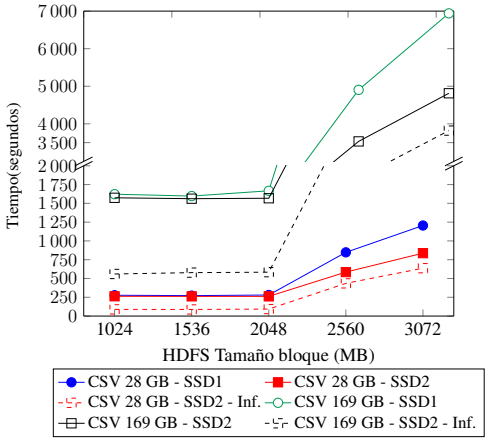


Figura 3.13: Tiempo de carga en HDFS de SSD a SSD

Configuración de YARN teniendo en cuenta el tamaño de bloque

Una vez que se ha calculado el máximo tamaño de bloque a utilizar, es posible comenzar con la configuración de cada una de las propiedades de YARN. Para poder realizar dicha configuración de manera sencilla y sistemática para cada una de las máquinas del clúster se ha desarrollado un algoritmo. Este algoritmo permite calcular los valores adecuados para cada una de las propiedades de YARN que permite distribuir los recursos asignados al servicio YARN NodeManager y así asegurar el máximo aprovechamiento de los recursos, es decir, el mejor rendimiento posible durante el proceso de carga. El algoritmo 4 muestra los distintos pasos a seguir y para ello es necesario aportar la información obtenida anteriormente. 1) La cantidad de RAM que va a tener disponible el servicio YARN NodeManager para crear contenedores. 2) El número de cores virtuales que va a tener el servicio YARN NodeManager para asignar a los distintos contenedores. Ambos valores dependerán de la propiedades de la máquina en la que vaya a ser ejecutado este servicio (sección 3.6). 3) El máximo tamaño de bloque a utilizar teniendo en cuenta las características de clúster. 4) El número de regiones de la tabla que van a ser servidas en la máquina en el que va a ejecutarse el servicio YARN NodeManager.

Primero se calcula la cantidad de RAM que va a ser asignada a los contenedores *map*, los contenedores tienen que ser capaces de almacenar los bloques que va leyendo del fichero y además los pares clave-valor obtenidos de la ejecución del trabajo *map*. Por tanto se va a asignar dos veces el tamaño máximo de bloque y la memoria que va a ser utilizada por la JVM, es un 20 % de la memoria del contenedor *map*(algoritmo 4, líneas 1 y 2). Dividiendo la cantidad de memoria total asignada al servicio YARN NodeManager (*ramTotal*) durante la distribución de los recursos de la máquina entre los servicios (sección 3.6, y teniendo en cuenta la memoria asignada a los contenedores *map*, obtenemos el número total de contenedores *map* que pueden ser ejecutados si solo tenemos en cuenta la memoria (algoritmo 4, línea 3). Por ejemplo supongamos que tenemos 24 GB de memoria total y 4 vcores asignados al servicio YARN NodeManager, como la cantidad de memoria RAM del contenedor *map* es 4 GB, en total se podrían ejecutar 24 / 4 = 6 contenedores *map*. Una vez tenemos el número de contenedores *map* podemos distribuir los vcores totales entre los contenedores *map* asegurando que como mínimo cada contenedor va a tener un vcore para poder ejecutarse (algoritmo 4, línea 4). Siguiendo con el ejemplo, los 4 vcores habría que dividirlos entre los 6 contenedores *map*, 4 / 6 = 0,667, como es menor que 1 que es el mínimo número de vcores que hay que asignar al contenedor, el número de contenedores *map* a ejecutar en paralelo será el mínimo del número de contenedores *map* teniendo en cuenta la memoria RAM y los vcores (algoritmo 4 líneas 5 y 6).

El número de regiones en la que está dividida la tabla va a definir el número de contenedores *reduce* a ejecutar en paralelo (algoritmo 4 línea 7). Hay que considerar que hay un contenedor más, él del *Application Master*, que se encuentra activo durante la vida de todo el proceso MapReduce por lo que tenemos que añadirlo al total de contenedores que se van a ejecutar en paralelo(algoritmo 4 línea 8). Con este dato, la cantidad de memoria RAM y el número de vcores que van a ser asignados a los contenedores *reduce*, dividiendo la cantidad de RAM total asignada al proceso YARN NodeManager entre el número de contenedores para obtener la cantidad de memoria a asignar al contenedor *reduce* (algoritmo 4, líneas 9 y 10). El número de vcores se

Algoritmo 4 Configurar recursos contenedores MapReduce

Require: ramTotal: cantidad de RAM asignada al servicio YARN NodeManager
Require: vcoresTotales: cantidad de cores virtuales asignados al servicio YARN NodeManager
Require: maxTamañoBloque: máximo tamaño de bloque adecuado al clúster
Require: #regionesNodo: cantidad de regiones de la tabla en cada máquina del clúster

1: ramMapCont. ← 2*maxTamañoBloque
2: ramJvmMapCont. ← ramMapCont * 0,8
3: #MapRamCont. ← ramTotal / ramMapCont.
4: vcoreMapCont. ← max (1, vcoresTotal/#MapRamCont.)
5: #MapVcoreCont. ← vcoresTotal / vcoreMapCont.
6: #MapCont. ← min(#MapRamCont., #MapVcoreCont.)
7: #ReduceCont. ← #regionesNodo
8: #Cont. ← #ReduceCont. + 1
9: ramReduceCont. ← ramTotal/#Cont.
10: ramJvmReduceCont. ← ramReduceCont * 0,8
11: vcoreReduceCont. ← vcoresTotal/#Cont. (min. 1 vcore)
12: ramAppMaCont. ← min(ramMapCont., ramReduceCont.)
13: vcoreAppMaCont ← min(vcoreMapCont., vcoreReduceCont.)

calcula dividiendo el total de vcores asignados al NodeManager entre el número de contenedores ejecutados en paralelo, siendo este valor como mínimo 1 (algoritmo 4 línea 11).

El contenedor para el *Application Master* puede ser configurado una vez que las propiedades para los contenedores *map* y el *reduce* han sido calculadas. Esto es debido a que el *Application Master* es un proceso que consume pocos recursos ya que su única función es gestionar el trabajo MapReduce. De esta manera la cantidad de memoria RAM asignada al *Application Master* es el mínimo asignado a los contenedores *map* y *reduce*. El número de vcores es el mínimo de los vcores asignados a los contenedores *map* y *reduce* (algoritmo 4, líneas 12 y 13).

Además de estas propiedades, hay otra propiedad importante que controla la inicialización de las tareas *reduce*. Esta propiedad llamada mapreduce.job.reduce.slowstart.completedmaps y define el porcentaje de tareas *map* que tienen que ser completadas antes de que se comiencen a lanzar las tareas *reduce*. Por omisión es de 0,05, es decir

que si han terminado al menos un 5 % de las tareas *map* en cuanto haya disponible recursos suficientes para crear un contenedor *reduce* éste va a ser creado. Este valor puede llegar a suponer un problema ya que si tenemos el caso de que haya que iterar más de una vez sobre los contenedores *map*, en cuanto una de las tareas *map* termine, el *Application Master* le va a dar prioridad a la creación de contenedores *reduce* retrasando la finalización de la tareas *map* y por tanto se retrasa la finalización del proceso de carga global. Para calcular el valor más adecuado para esta propiedad se han definido las siguientes dos fórmulas que tienen en cuenta el tamaño del fichero, el tamaño de bloque utilizando en la carga del fichero a HDFS y la cantidad de contenedores *map* que se pueden ejecutar en paralelo. Con la fórmula 3.2, se va a obtener el número de iteraciones de contenedores *map* (#MapIteraciones). Por ejemplo, para un fichero de 379 GB que es cargado con un tamaño de bloque de 1631 MB, el número de bloques generados por HDFS es 238. Teniendo en cuenta que un escenario en el que se pueden ejecutar en paralelo hasta 119 contenedores *map*, el número de iteraciones de tareas *map* que hay que realizar son 238 / 119 = 2. Con este valor, la fórmula 3.3 va a calcular el porcentaje de tareas *map* que tienen que haber terminado antes de que se cree el primer contenedor *reduce* de la ejecución. Siguiendo con el mismo ejemplo, el porcentaje obtenido será 1 - (1 / #MapIteraciones) = 1 - (1 / 2) = 0.5. Es decir, al menos la mitad de las tareas *map* tienen que haber terminado antes de que el primer *reduce* comience para así poder asegurar un buen rendimiento.

$$\#MapIteraciones = \frac{\#BloquesCSV}{\#MapParalelo} \tag{3.2}$$

$$reduce.slowstart.completedmap = 1 - \frac{1}{\#MapIteraciones} \tag{3.3}$$

En la tabla 3.3 se muestra en resumen de los valores de cada una de las propiedades de YARN para que se aprovechen los recursos que de la máquina en la que se ejecute el servicio YARN NodeManager.

Tabla 3.3: Configuración del servicio YARN_NM aprovechando los recursos asignados a YARN

Propiedad	Valor
yarn.scheduler.minimum-allocation-mb	min(mapreduce.map.memory.mb, mapreduce.reduce.memory.mb)
yarn.scheduler.maximum-allocation-mb	ramTotal
yarn.nodemanager.resource.memory-mb	ramTotal
yarn.nodemanager.resource.cpu-vcores	vcoresTotales
mapreduce.map.memory.mb	maxTamañoBloque * 2
mapreduce.map.java.opts	mapRAM * 0,8
mapreduce.map.cpu.vcores	vcoresTotales/#mapContainers
mapreduce.reduce.memory.mb	ramTotal /#reduceContainers+1
mapreduce.reduce.java.opts	reduceRAM * 0,8
mapreduce.reduce.cpu.vcores	vcoresTotales/#reduceContainers+1
yarn.app.mapreduce.am.resource.mb	min(mapreduce.map.memory.mb, mapreduce.reduce.memory.mb)
yarn.app.mapreduce.am.command-opts	min(mapreduce.map.memory.mb, mapreduce.reduce.memory.mb) * 0.8
mapreduce.job.reduce.slowstart. completedmaps	Ecuaciones 3.2 y 3.3

3.8. Ubicación de las regiones

HBase almacena sus datos en HDFS en ficheros HFiles, por omisión el nivel de replicación de los ficheros HFile es 3. Para cada región y familia de columnas de la tabla se crea un HFile encargado de almacenar dichos datos. La primera réplica de los HFiles siempre se encuentra ubicada en la misma máquina donde se encuentra el HBase region server que sirve dicha región. Cuando una región se mueve de un HBase region server a otro que se encuentre en otra máquina, los datos permanecen en la misma máquina que estaban en origen. Esto se traduce en un aumento de la latencia en el acceso de los datos hasta que finalmente el fichero HFile sea migrado, mediante el comando *compact* de HBase, desde ese HDFS DataNode origen hasta el que se encuentre en la misma máquina del nuevo HBase region server que esté sirviendo la región movida.

En caso de que un corte de luz detenga los servicios del clúster, al volver a iniciar HBase, éste no sabe en qué máquina estaban los datos antes de que el sistema se detuviese, por ello comienza a asignar las regiones sin conocer en qué máquina se encontraban los datos almacenados. HBase recomienda ejecutar el proceso *compact* de tal manera que los datos se muevan a la máquina en la que se ha registrado la región al

iniciar el sistema. Esto se debe a que si el clúster tiene un número de máquinas superior al nivel de replicación de los bloques de los ficheros, hay máquinas que no tienen almacenadas réplicas de los bloques. Este proceso es muy costoso ya que supone la copia uno a uno de los HFiles desde la máquina inicial a la nueva máquina.

Para evitar este inconveniente, se ha implementado e incorporado un proceso al final del proceso de carga de tal manera que se almacena permanentemente en ZooKeeper en qué HBase region server se encuentra cada una de las regiones de la tabla cargada, que responde con las mismas máquinas en donde se han guardado los HFiles de las regiones en la carga. Si el sistema es reiniciado, se ejecuta otro proceso implementado que accede a ZooKeeper y comienza a mover las regiones de manera lógica al HBase region server que estaba sirviendo la región antes de producirse el corte de luz. De esta manera, se asegura de que el sistema está disponible mucho más rápidamente ya que no se migran los datos entre los nodos del sistema. El proceso *move* (sección 3.5) se realiza de manera lógica para HBase, es decir que HBase registra la región en la máquina que se indique sin mover los datos de una máquina a otra y en cambio el proceso de migración (*compact*) envía todos los datos por red desde una máquina a la otra.

3.9. Evaluación del rendimiento

Para llevar acabo la evaluación de rendimiento de las distintas herramientas y algoritmos propuestos en este capítulo se han utilizado tres escenarios diferentes: 1) Clúster AMD, 2) Clúster Intel XEON y 3) una máquina manycore, el Bullion Sequana S800. El clúster AMD consiste en 11 nodos homogéneos conectados a través de una red Ethernet de un 1 Gb. Cada uno de los nodos está equipado con 4 CPU o sockets con un procesador AMD Opteron 6376 @ 2.3GHz y cada uno tiene 8 cores (16 vcores), es decir un total de 64 virtual cores por nodo. Además, tiene 128 GB de memoria RAM dividida en 16 módulos de 8 GB cada una. Todos los nodos tienen conectados directamente dos discos, un SSD (Intel SD3500 480GB) y un HDD con 2TB de capacidad. El clúster Intel XEON está formado por 11 máquinas cada una de ellas equipada con

2 CPU o sockets con un procesador Intel XEON E5-2620 v3 con 12 cores, es decir un total de 24 cores virtuales. Cada máquina tiene 128 GB de memoria RAM repartida en 8 módulos, cada una de ellas contiene un tarjeta RAM de 16 GB. Directamente conectados tiene 3 discos, dos son SSD: 1) Intel SD3510 480GB, 2) Intel SD3520 y el tercero es un disco HDD de 4 TB de capacidad. Todos ellos están conectados a través de una red Ethernet de 1 Gbit. El Bullion Sequana S800 es una máquina equipada con 4TB de memoria RAM dividida en 8 ranuras, cada una de ellas con una tarjeta de 512 GB de memoria. Esta máquina tiene 4 módulos con dos sockets cada uno y cada socket contiene un Intel(R) Xeon(R) Platinum 8158 CPU @ 3.00GHz con 12 cores (24 vcores) dando lugar a un total de 192 cores virtuales. Cada uno de los sockets pares tiene directamente conectado 3 discos al bus PCI, resultando un total de 12 discos SSD de 512 GB cada uno.

Dado que la distribución de los servicios en los distintos clústers y la asignación de recursos a cada uno de ellos tiene un gran impacto en la configuración de las herramientas necesarias para realizar la carga de los datos, en esta sección se va a demostrar el funcionamiento del algoritmo 3 presentado en la Sección 3.6 de este capítulo. Seguidamente se presentarán las distintas evaluaciones del rendimiento del proceso de carga. Se ha evaluado: 1) El impacto del tamaño de bloque en el proceso de carga. 2) Tiempo de carga utilizando las distintas configuraciones de YARN presentadas en las Sección 3.7 y 3.7.0.1. Y 3) La escalabilidad del las herramientas de Pre-Split e Inyección de carga presentadas en las secciones 3.5 y 3.4 en los clúster AMD, XEON y con el Bullion Sequana S800.

Tabla 3.4: Arquitectura de las máquinas utilizadas durante la evaluación

	AMD	XEON	Bullion
nodos	11	11	1
red	1Gb	1Gb	-
sockets	4 módulos x 2 sockets	2 módulos x 1 socket	4 módulos x 2 sockets
vcores	64	24	192
RAM (GB)	128	128	4096
discos	1 HDD + 2 SSD	1 HDD + 2 SSD	12 SSD
procesador	AMD Opteron 6376 @ 2.3GHz	Intel XEON E5-2620 v3	Intel(R) Xeon(R) Platinum 8158 CPU @ 3.00GHz

3.9.1. Configuración de los distintos clústers

Tanto el clúster AMD como el XEON tienen 11 nodos por lo que la distribución de servicios en las máquinas se realiza de la misma manera. Uno de los nodos va a ejecutar los procesos máster de HDFS, HBase y YARN, a este nodo lo vamos a llamar *Nodo de control*. Este nodo va a tener los siguientes procesos en ejecución ZooKeeper, HDFS NameNode, YARN ResourceManager y HBase Master. El resto de nodos va a ejecutar los siguientes servicios: un servicio HDFS DataNode, un servicio YARN NodeManager y 4 servicios HBase region server. En la parte a) de la figura 3.7 se muestra el despliegue de los distintos servicios en ambos clústers.

Antes de comenzar con la evaluación del rendimiento del inyector de datos masivo se han cargado las 9 tablas del benchmark TPC-C [41]. El tamaño de la base de datos se define por número de warehouses (WHs) de tal manera que el tamaño de cada una de las tablas se encuentra dimensionado por este valor. Por ejemplo, una base de datos de 1 warehouse equivale a un tamaño de 65 MB, donde la tabla más pequeña, warehouse tiene una tupla y la tabla más grande Order_Line tiene 255000 tuplas. Para determinar el número de instancias HBase region server que pueden ser colocadas en una máquina AMD se ha realizado una evaluación cargando las 9 tablas del benchmark TPC-C y ejecutando el benchmark durante 20 minutos con 1,2,3,4 y 5 HBase region servers en la misma máquina utilizando bases de datos de 50, 100, 200, 300, 400 y 500 warehouses. En la figura 3.14 se muestran los resultados de la evaluación (Throughput, Latencia y consumo de CPU), de la misma se obtiene que el mejor rendimiento con la configuración de 4 instancias HBase region server con la base de datos de 300 warehouses con una media de 3733 transacciones por minuto ejecutadas, latencia de 266 milisegundos y un consumo de CPU de 68 %.

Como el objetivo de esta evaluación es estudiar el rendimiento del inyector de carga y no el rendimiento de la base de datos se van a utilizar distintos tamaños de bases de datos 1000, 3000, 6000, 12000, 18000 y 36000 warehouses para las cuales los tamaños de los ficheros CSV de todas las tablas que van a ser cargadas son 65 GB, 169 GB, 390 GB ,781 GB, 1230 GB y 3788 GB. Además con el objetivo de comparar resultados en

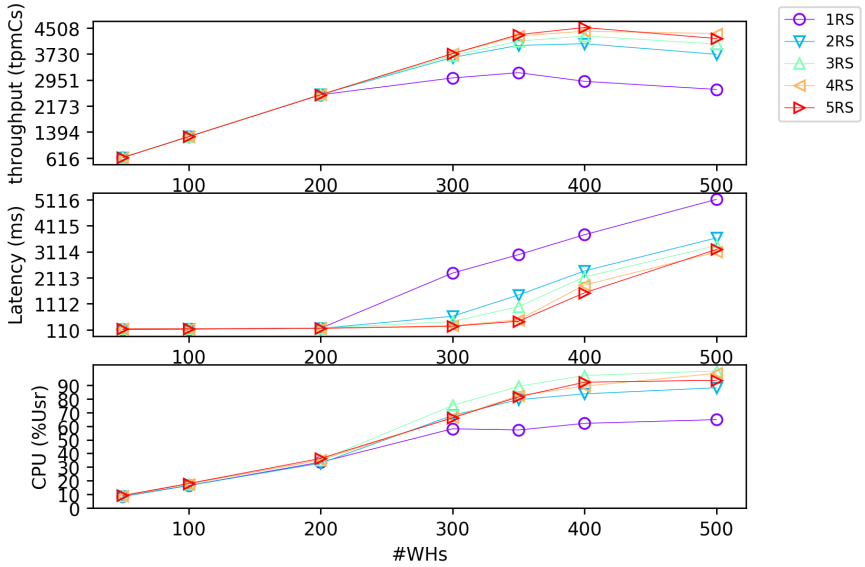


Figura 3.14: Evaluación benchmark TPC-C en un nodo AMD

los clústers AMD y XEON se ha utilizado el mismo número de HBase region servers (4) y en ambos escenarios se ha utilizado la comunicación entre máquinas por medio de la conexión de red ethernet a 1 GB. La distribución de los recursos en la máquinas de ambos clúters se presentan en las secciones 3.9.1.1 y 3.9.1.2, respectivamente.

En el caso del Bullion Sequana S800, al tratarse de una única máquina todos los servicios van a ser ejecutados en ella. Para aislar los servicios que están ejecutando en el *Nodo de control* de los otros dos clústers, en este caso los servicios se desplegarán en una de las unidades NUMA y el resto de servicios en las unidades NUMA restantes (parte b), figura 3.7). Con el objetivo de desplegar el mayor número de instancias HBase region server se ha decidido colocar una instancia por core físico y asignarle la cantidad de memoria RAM que le pertenece a cada core, 36 HBase region servers cada uno con 42 GB de RAM. Esta distribución se muestra en detalle en la sección 3.9.1.3.

3.9.1.1. Configuración del clúster AMD

Los *nodos de datos* del clúster AMD son configurados utilizando el algoritmo 3. En la Sección 3.6 se ha explicado la aplicación de dicho algoritmo utilizando la arquitec-

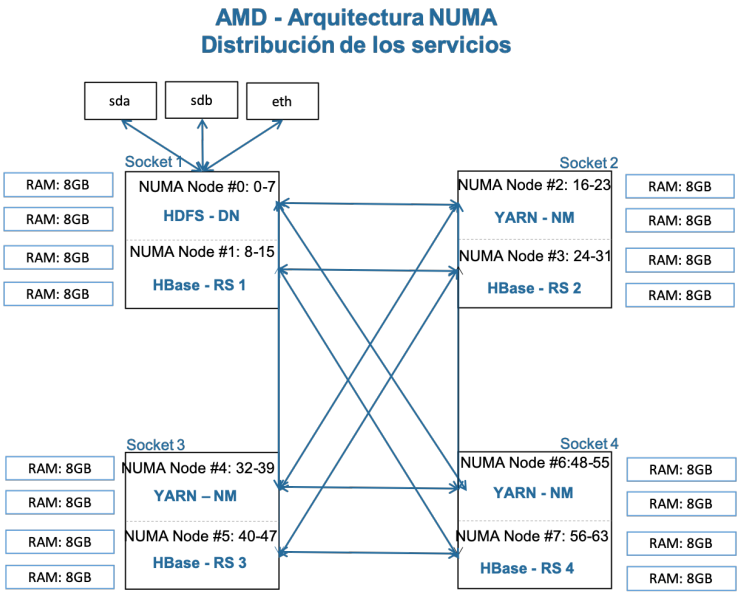


Figura 3.15: Distribución de los servicios en un nodo AMD.

tura de las máquinas AMD (figura 3.8). Como resultado de la ejecución del algoritmo la distribución de los recursos de las máquinas del clúster AMD la de la figura 3.15 donde el servicio HDFS-DataNode es asignado en el nodo NUMA #0 ya que tiene directamente conectados los discos en los que va a almacenar los datos. Los nodos NUMA #2, #4 y #6, son asignados al servicio YARN-NodeManager debido a su baja latencia de acceso al nodo NUMA #0, lo cual supone que se van a asignar 48 GB y 24 vcores a dicho proceso. Y el resto de nodos NUMA #1, #3, #5 y #7 van a ser asignados a los HBase region servers 1, 2 ,3 y 4 respectivamente, de tal manera que cada HBase region server va a tener 16 GB de memoria RAM y 8 vcores asignados.

A continuación se configuran los recursos de los contenedores de YARN, para ello se ejecuta el algoritmo 4. Tras realizar la distribución de los recursos de los cuales 48 GB y 24 cores virtuales han sido asignados al servicio YARN NodeManager (filas 1 a 4, tabla 3.5), se tiene los valores de las variables que requiere el algoritmo: 1) ramTotal = 49152 MB, 2) vcoresTotales = 24 vcores, 3) maxTamañoBloque, en este caso definido a 2048 MB, calculado en la Sección 3.7.0.1 y 4) #regionesNodo = 4 debido a que vamos a crear una región por HBase region server del nodo. Primero se

asigna la memoria de los contenedores *map* con los valores 4096 MB y -Xmx3276m (algoritmo 4, líneas 1 y 2). A continuación se calcula el número de contenedores *map* mediante la división de la memoria RAM total entre la asignada a los contenedores, 49125 MB / 4096 MB = 12 contenedores *map* (algoritmo 4, línea 3) y de ahí se obtiene la cantidad de vcores virtuales que se van a asignar a cada contenedor *map*, 24 vcores / 12 contenedores = 2 vcores. Se comprueba si el número de contenedores *map* distribuyendo la memoria corresponde con los obtenidos tras distribuir los vcores (algoritmo 4, líneas 4 a 6) y así se finalizaría la configuración de los contenedores *map*, filas 5 a 7 de la tabla 3.5. Hay que tener en cuenta que uno de esos contenedores va a ser utilizado por el proceso *Application Master* cuya configuración se calcula en último lugar, por lo tanto habría un total de 11 contenedores *map* en caso de que se ejecute el contenedor *Application Master* en dicho nodo, en caso contrario serán 12 los contenedores *map*.

En cuanto a la configuración de los contenedores *reduce*, se comienza con el cálculo del número de contenedores totales (contenedores *reduce* + contenedor *Application Master*) que se tienen que ejecutar en paralelo (algoritmo 4, líneas 7 y 8), #regiones-Nodo + 1 = 5 contenedores. Se calcula la cantidad de RAM a asignar a cada contenedor dividiendo la memoria RAM asignada al servicio YARN NodeManager entre el número de contenedores, 49125 MB / 5 contenedores = 9830 MB (9830 * 0,8 = 7864 MB memoria asignada a la JVM de los contenedores *reduce*). Y finalmente, se calcula la cantidad de cores virtuales repartiendo los vcores asignados al proceso YARN NodeManager entre el número de contenedores, 24 vcores / 5 contenedores = 4 vcores, (líneas 9 a 11 del algoritmo 4 y filas 8 a 10 de la tabla 3.5). La configuración para el contenedor *Application Master* se calcula como el mínimo valor configurado para el contenedor *map* o el contenedor *reduce* (líneas 12 y 13, algoritmo 4) y filas 11 y 12 de la tabla 3.5. La propiedad mapreduce.job.reduce.slowstart.completedmaps va a ser modificada teniendo en cuenta el tamaño de datos cargados en cada uno de los experimentos y se especificará en cada una de las evaluaciones el valor asignado.

Tabla 3.5: Configuración YARN nodo AMD

Propiedad	Cómo se calcula	Valor
yarn.scheduler.minimum-allocation-mb	min(mapreduce.map.memory.mb, mapreduce.reduce.memory.mb)	min(4096, 9830) = 4096
yarn.scheduler.maximum-allocation-mb	ramTotal	49152 MB
yarn.nodemanager.resource.memory-mb	ramTotal	49152 MB
yarn.nodemanager.resource.cpu-vcores	vcoreTotales	24
mapreduce.map.memory.mb	maxTamañoBloque * 2	4096 MB
mapreduce.map.java.opts	mapRAM * 0,8	-Xmx3276m
mapreduce.map.cpu.vcores	vcoreTotales/#mapContenedores	24/(49152/4096) = 2
mapreduce.reduce.memory.mb	ramTotal /#reduceContenedores+1	49152/(4+1)= 9830 MB
mapreduce.reduce.java.opts	reduceRAM * 0,8	9830*0,8 = -Xmx7864m
mapreduce.reduce.cpu.vcores	vcoreTotales/#reduceContainers+1	24/(4+1) = 4
yarn.app.mapreduce.am.resource.mb	min(mapreduce.map.memory.mb, mapreduce.reduce.memory.mb)	min(4096, 9830) = 4096 MB
yarn.app.mapreduce.am.command-opts	min(mapreduce.map.memory.mb, mapreduce.reduce.memory.mb) * 0.8	Xmx3276m

3.9.1.2. Configuración del clúster XEON

La distribución de los recursos en los *nodos de datos* del clúster XEON ha sido mostrada como ejemplo del escenario tipo dos en el algoritmo 3, debido a que el número de servicos a desplegar es inferior al número de nodos NUMA. El resultado es la distribución mostrada en la figura 3.16. En dicha figura se puede ver que los servicios HDFS-DataNode y YARN-NodeManager se encuentran aislados en el nodo NUMA #0, dicho nodo tiene directamente conectados los discos en los que el servicio HDFS-DataNode va a guardar los datos de las tablas por lo que se le asignaran los cores virtuales 0 y 12 con su memoria RAM y el resto de cores virtuales (1-5, 13-17) van a ser asignados al proceso YARN NodeManager con un total de 53 GB RAM, que es la que corresponde a los 10 vcores asignados a dicho proceso. Por otro lado, los cuatro HBase region servers se ejecutarán en la unidad NUMA con identificador #1, esto significa que los 12 vcores y los 64 GB de memoria RAM van a ser distribuidos entre los 4 HBase region servers, por lo que cada uno va a tener asignados 16 GB de memoria.

Una vez distribuidos los recursos del nodo entre los distintos servicios se configura las propiedades de servicio YARN NodeManager al que se le han asignado 53 GB RAM y 10 cores virtuales (filas 1 a 4, tabla 3.6). Se comienza por la asignación de memoria RAM a los contenedores *map* cuyo valor es el doble del máximo tamaño de

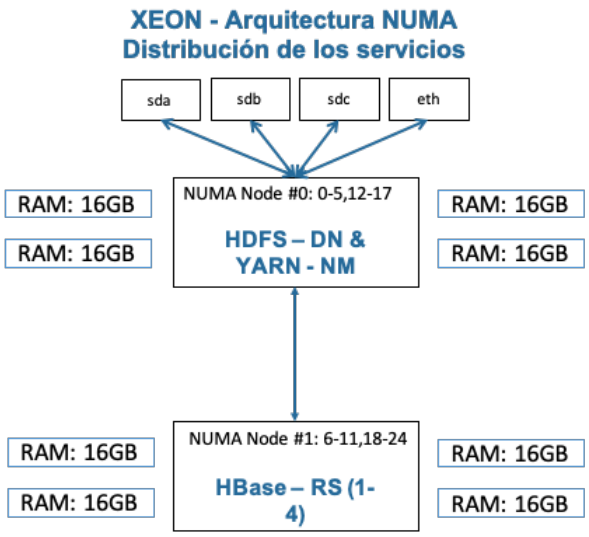


Figura 3.16: Distribution of services in XEON NUMA Architecture.

bloque que va a utilizar en este despliegue, en este caso son 2 GB tal y 4096 MB de memoria RAM (3276 MB asignados a la máquina virtual de Java). Una vez se conoce la memoria RAM asignada al proceso YARN NodeManager y la memoria de los contenedores *map*, se puede calcular el número total de contenedores *map* que pueden ejecutarse en paralelo: 54272 MB ramTotal / 4096 MB ramMapCont. = 13 contenedores. A continuación se distribuyen los vcores totales entre los distintos contenedores y se vuelve a calcular el número de contenedores *map*: 10 vcoresTotales / 13 contenedores = 0,76. Dado que como mínimo tiene que ser asignado un vcore a cada contenedor, solo es posible ejecutar 10 contenedores *map* en paralelo (líneas 1 a 6 del algoritmo 4 y filas 5 a 7 de la tabla 3.6). En el nodo en el que se ejecute el contenedor principal *Application Master* el número total de contenedores *map* se verá reducido a 9, el décimo será el que esté ocupado por dicho servicio.

Para configurar los contenedores *reduce*, primero se calcula el número de contenedores totales que se pueden ejecutar en paralelo. Tiene que haber tantos contenedores *reduce* como regiones de la tabla servidas en el nodo a configurar, es decir 4. A estos contenedores hay que añadirle uno que va a ser ocupado por el servicio *Application Master* por lo que habrá 5 contenedores ejecutados en paralelo. A continuación se cal-

cula la cantidad de RAM que se asigna a cada contenedor *reduce* 54272 MB ramTotal/ 5 contenedores = 10240 MB. El número de cores virtuales a asignar también se calcula de la misma manera, repartiendo los cores virtuales entre los contenedores: 10 vcores-Totales / 5 contendores = 2 vcores por contenedor (líneas 7 a 11, algoritmo 4 y filas 8 a 10, tabla 3.6). Una vez configurados los contenedores *map* y *reduce* se configura el contenedor *Application Master* asignando la memoria mediante min(ramMapCont, ramReduceCont) = min (4096, 10854) (líneas 12 y 13, algoritmo 4 y filas 11 y 12, tabla 3.6).

Tabla 3.6: Configuración YARN nodo XEON

Propiedad	Cómo se calcula	Valor
yarn.scheduler.minimum-allocation-mb	min(mapreduce.map.memory.mb, mapreduce.reduce.memory.mb)	min(4096, 10854) = 4096 MB
yarn.scheduler.maximum-allocation-mb	totalRAM	54272 MB
yarn.nodemanager.resource.memory-mb	totalRAM	54272 MB
yarn.nodemanager.resource.cpu-vcores	totalVirtualCores	10
mapreduce.map.memory.mb	maxTamañoBloque * 2	4096 MB
mapreduce.map.java.opts	mapRAM * 0,8	-Xmx3276m
mapreduce.map.cpu.vcores	totalVirtualCores/#mapContenedores	10/(54272/4096) = 1
mapreduce.reduce.memory.mb	TotalRAM /#reduceContenedores+1	54272/(4+1)= 10854 MB
mapreduce.reduce.java.opts	reduceRAM * 0,8	10854*0,8 = -Xmx8683m
mapreduce.reduce.cpu.vcores	totalVirtualCores/#reduceContainers+1	10/(4+1) = 2
yarn.app.mapreduce.am.resource.mb	min(mapreduce.map.memory.mb, mapreduce.reduce.memory.mb)	min(4096, 9830) = 4096 MB
yarn.app.mapreduce.am.command-opts	min(mapreduce.map.memory.mb, mapreduce.reduce.memory.mb) * 0.8	Xmx3276m

3.9.1.3. Configuración del Bullion Sequana S800

El Bullion Sequana S800 es un súper ordenador con arquitectura NUMA (figura 3.17). Está compuesto de 4 módulos cada uno de ellos equipado con 2 procesadores (nodos NUMA), cada procesador tiene 24 cores virtuales con Hyper threading habilitado que produce pares de cores virtuales (0,96 1,97, ...) los cuales comparten thread. Cada uno de los procesadores tiene directamente conectado una tarjeta de memoria de 512 GB de RAM. Los discos están directamente conectados a las unidades NUMA impares, lo que significa que los procesos ejecutándose en estas unidades van a tener un acceso más rápido a los discos. En la figura 3.18, se muestran las conexiones con menos latencia entre las distintas unidades. Por ejemplo, la distancia a la memoria de

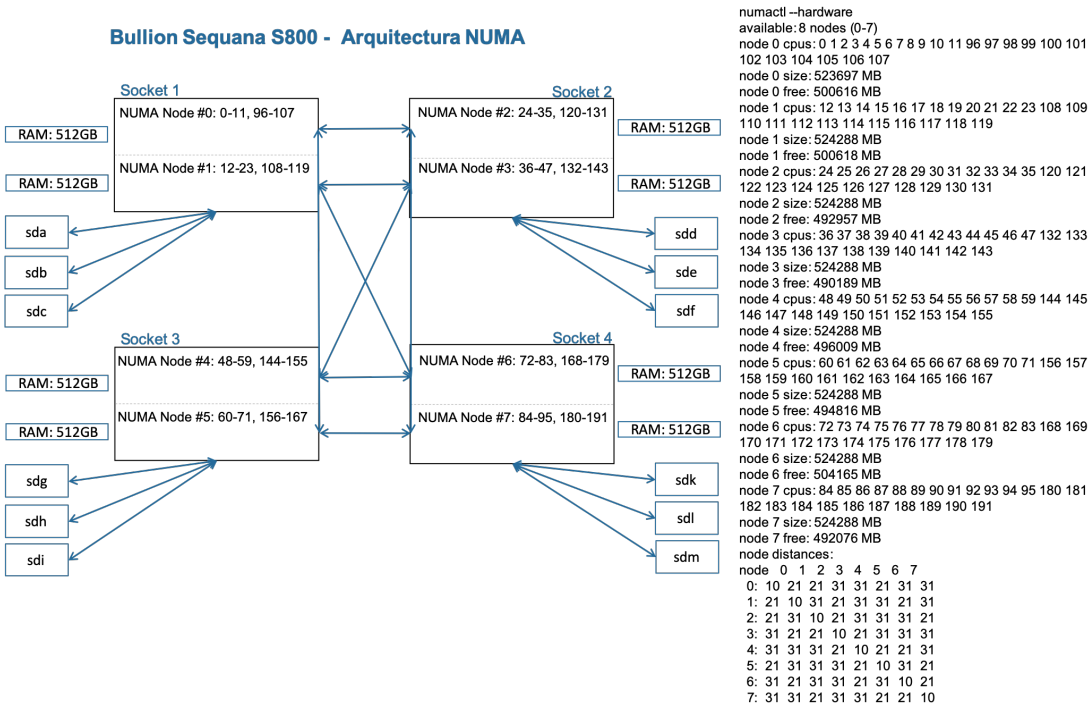


Figura 3.17: Arquitectura NUMA de los nodos BSequana S800

la unidad NUMA con identificador #0 es menor desde las unidades NUMA #1, #2 y #5, etc.

Para distribuir los servicios en esta máquina, tal y cómo se ha explicado en la Sección 3.6 los servicios HDFS NameNode, ZooKeeper, YARN ResourceManager y HBase Master se ejecutarán en la unidad NUMA #0, ya que estos procesos no tienen un excesivo acceso a disco. El resto de servicios van a ser distribuidos entre las unidades NUMA restantes (#1, #2, #3, #4, #5, #6 y #7). Una vez definidos los nodos NUMA en los que se van a desplegar los servicios HDFS DataNode, YARN NodeManager y HBase region server, se aplica el algoritmo 3 para la distribución de los recursos. Primero se asigna el nodo NUMA y la memoria RAM correspondiente al servicio HDFS NodeManager teniendo en cuenta la localización de los discos dentro de la arquitectura NUMA; se escoge entre la lista de nodos NUMA con disco el nodo NUMA #1 y su memoria correspondiente que es 512 GB. Ya que el servicio YARN NodeManager tiene que estar junto a los nodos NUMA que tengan el menor número de saltos de

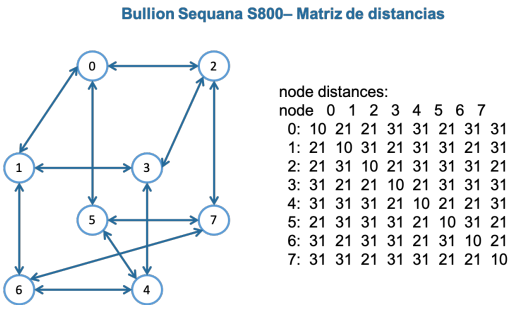


Figura 3.18: Matriz de distancia NUMA en los nodos BSequana S800

acceso a la memoria del nodo en el que se va a desplegar el servicio HDFS DataNode, el servicio YARN NodeManager va a ser asignando a los nodos NUMA #3, #5 y #7 de tal manera que este servicio va a tener asignados 1,5 TB de memoria RAM. El resto de nodos NUMA (#2, #4 y #6) van a ser asignados a los procesos HBase region server. En este caso el número de HBase region server es de 36 en total (uno por core físico), 12 por unidad NUMA, asignando a cada uno 2 vcores y 42 GB de memora RAM. Esta configuración se ha elegido de tal manera que permita tener el máximo número de servicios HBase region server y así aprovechar al máximo los recursos de esta máquina. La figura 3.19, muestra cómo quedaría la distribución de los recursos.

Para finalizar se configuran las propiedades del servicio YARN NodeManager teniendo en cuenta la memoria RAM asignada, ramTotal = 1,5 TB y el número de cores vcoresTotales = 72 vcores, las filas 1 a 4 de la tabla 3.7 muestran la asignación de estos valores en las propiedades correspondientes. Primero se asignan las propiedades de los contenedores *map*, líneas 1 a 6 del algoritmo 4. La memoria asignada es el doble del tamaño máximo de bloque, 2048 MB por lo que el valor asignado a la propiedad mapreduce.map.memory.mb es 4096 MB, y se calcula el número de contenedores que se podrían lanzar en paralelo teniendo en cuenta la memoria RAM: 1572864 MB ramTotal / 4096 MB ramMapCont = 384 contenedores. Debido a que este número es superior a la cantidad de cores virtuales asignados al servicio, el número de contenedores se verá limitado a 72, cada contenedor tendrá asignado un core virtual. Uno de los 72 contenedores va a ser utilizado por el proceso *Application Master*, como máximo se podrán ejecutar en paralelo 71 contenedores *map*. Las filas 5 a 7 de la tabla 3.7

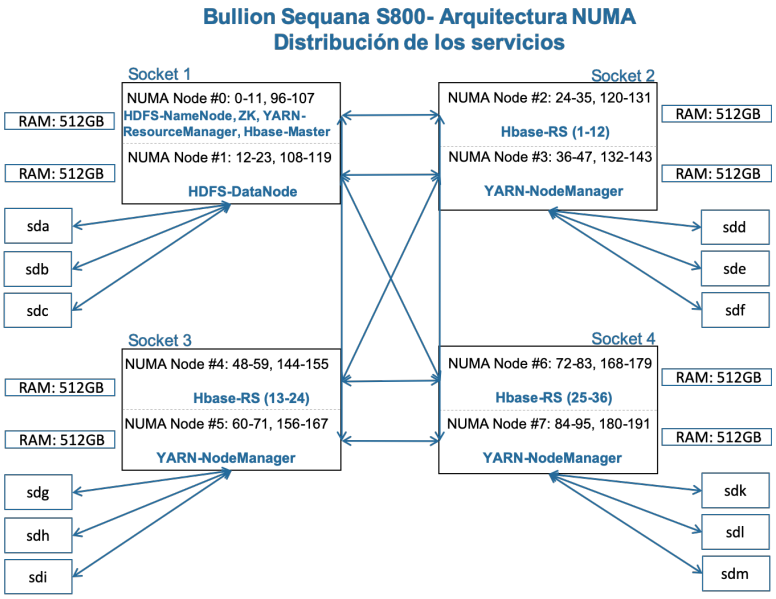


Figura 3.19: Distribution of services in BSequana NUMA Architecture.

muestran cómo quedan configuradas las propiedades de los contenedores *map*.

A continuación se calcula el número de contenedores totales, contenedores *reduce* más el contenedor para el *Application Master*. Como tiene que haber un contenedor *reduce* por región en total tiene que haber 36 contenedores *reduce* + 1 contenedor *Application Master* = 37 contenedores, líneas 7 y 8 del algoritmo 4. Se calcula la cantidad de RAM y número de cores virtuales dividiendo el total de memoria RAM y cores virtuales entre los contenedores respectivamente: ramReduceCont = 1572864 MB ramTotal / 37 contenedores = 42509 MB de RAM y vcoreReduceCont = 72 vcoresTotal / 37 contenedores = 1 vcore por contenedor (filas 8 a 10 de la tabla 3.7). Para finalizar se configura las propiedades del contenedor *Application Master* (líneas 9 y 10 del algoritmo 4): ramAppMaCont = min(ramMapCont, ramReduceCont) y vcoresAppMaCont = min(vcoresMapCont, vcoresReduceCont), filas 11 y 13 de la tabla 3.7.

Resumen de la configuración YARN de los entornos utilizados

La tabla 3.8 muestra un resumen del número de contenedores *map* y *reduce* que

Tabla 3.7: Configuración YARN Bullion Sequana S800

Propiedad	Cálculo	Valor
yarn.scheduler.minimum-allocation-mb	min(mapreduce.map.memory.mb, mapreduce.reduce.memory.mb)	min(4096, 42509) = 4096 MB
yarn.scheduler.maximum-allocation-mb	totalRAM	15728864 MB
yarn.nodemanager.resource.memory-mb	totalRAM	15728864 MB
yarn.nodemanager.resource.cpu-vcores	totalVirtualCores	72
mapreduce.map.memory.mb	maxTamañoBloque * 2	4096 MB
mapreduce.map.java.opts	mapRAM * 0,8	-Xmx3276m
mapreduce.map.cpu.vcores	totalVirtualCores/#mapContenedores	72/(15728864/4096) = 1
mapreduce.reduce.memory.mb	TotalRAM /#reduceContenedores+1	15728864/(36+1)= 42509 MB
mapreduce.reduce.java.opts	reduceRAM * 0,8	42509*0,8 = -Xmx34007m
mapreduce.reduce.cpu.vcores	totalVirtualCores/#reduceContainers+1	72/(36+1) = 2
yarn.app.mapreduce.am.resource.mb	min(mapreduce.map.memory.mb, mapreduce.reduce.memory.mb)	min(4096, 9830) = 4096 MB
yarn.app.mapreduce.am.command-opts	min(mapreduce.map.memory.mb, mapreduce.reduce.memory.mb) * 0.8	Xmx3276m

se pueden ejecutar en paralelo en cada uno de los 3 entornos que se van a utilizar en las evaluaciones teniendo en cuenta la configuración de YARN implementada. De esta manera, con la configuración predefinida de YARN tanto en el clúster AMD como en el XEON se pueden ejecutar 78 contenedores *map* y *reduce* en paralelo, siendo 6 en el caso del Bullion Sequana S800. Con la configuración obtenida mediante la herramienta de HortonWorks yarn-util.py obtenemos 39 contenedores *map* y *reduce* en el clúster AMD, 59 en el XEON y 21 en el Bullion Sequana S800. El mayor paralelismo se consigue utilizando la configuración presentada en este capítulo siendo de 119 contenedores *map* y 59 contenedores *reduce* en el clúster AMD, 99 contenedores *map* y 49 contenedores *reduce* en el clúster XEON y 71 para los contenedores *map* y *reduce* en el Bullion Sequana S800.

Tabla 3.8: Contenedores YARN en AMD, XEON y Bullion

	Predeterminada		yarn-util.py		Inyector Carga	
	Map	Reduce	Map	Reduce	Map	Reduce
AMD	78	78	39	39	119	59
XEON	78	78	59	59	99	49
Bullion	6	6	21	21	71	71

Observando el número de contenedores tanto *map* como *reduce* obtenidos con las distintas configuraciones YARN. La configuración obtenida mediante las contribuciones presentadas en las secciones 3.6 y 3.7 permite obtener un mayor paralelismo y

aprovechar los recursos de la mejor manera posible. A continuación en las siguientes evaluaciones se va a demostrar entre otras cosas la mejora del rendimiento en el inyector de carga obtenida mediante la configuración propuesta.

3.9.2. Impacto del tamaño de bloque en el tiempo de carga

Para demostrar la importancia que tiene el tamaño del bloque en el rendimiento del proceso de carga, se va a utilizar dos tamaños diferentes del fichero para cargar la misma tabla en HBase, un fichero de 28 GB (28672 MB) con 100M de filas y un fichero de mayor tamaño de 169 GB (173056 MB) con 300M de filas. Como se ha mencionado con anterioridad el tamaño de los bloques define el número total de tareas *map* a ejecutar durante el proceso de carga. Para ver el impacto, se han utilizado los clúster AMD y XEON con la configuración de YARN definida por la herramienta *yarn-util.py*, de tal manera que hay un máximo paralelismo de 39 tareas *map* en el clúster AMD y 59 en el clúster XEON. En total cada tabla se ha dividido en 40 regiones utilizando el Pre-Split antes de comenzar con el proceso de carga. Se va a variar el tamaño de bloque desde 128 MB hasta 2048 MB con ambos ficheros y en cada uno de los clústers.

En la tabla 3.9 se muestra para cada uno de los ficheros, el número de iteraciones de tareas *map* dependiendo del clúster en el que se realiza la carga y el número total de tareas dependiendo del tamaño de bloque del fichero en HDFS. Por ejemplo, con un tamaño de bloque de 128 MB con el fichero de 28 GB se crean 224 bloques en HDFS por lo tanto hay un total de 224 tareas *map*. Teniendo en cuenta que el clúster AMD tiene un paralelismo de 39 tareas *map* por iteración, se van a realizar 6 iteraciones (ejecución de 39 tareas *map* en paralelo) y en el clúster XEON 4 iteraciones ya que el paralelismo de las tareas *map* es de 59 por iteración. De esta misma forma se puede conocer el número de iteraciones y tareas *map* para el resto de procesos de carga realizados en esta evaluación.

La figura 3.20 presenta el tiempos de carga en segundos para cada una de las ejecuciones realizadas, cada ejecución se ha realizado 3 veces y se ha hecho la media de los

Tabla 3.9: Número de iteraciones y tareas Map según el tamaño de bloque

Tamaño Fichero	28 GB = 28672 MB			169 GB = 173056 MB		
Tamaño Bloque	AMD Iteraciones	XEON Iteraciones	Maps	AMD Iteraciones	XEON Iteraciones	Maps
128 MB	6	4	224	35	23	1352
256 MB	3	2	112	18	12	676
512 MB	2	1	56	9	6	338
1024 MB	1	1	28	5	3	169
1536 MB	1	1	19	3	2	113
2048 MB	1	1	8	3	2	85

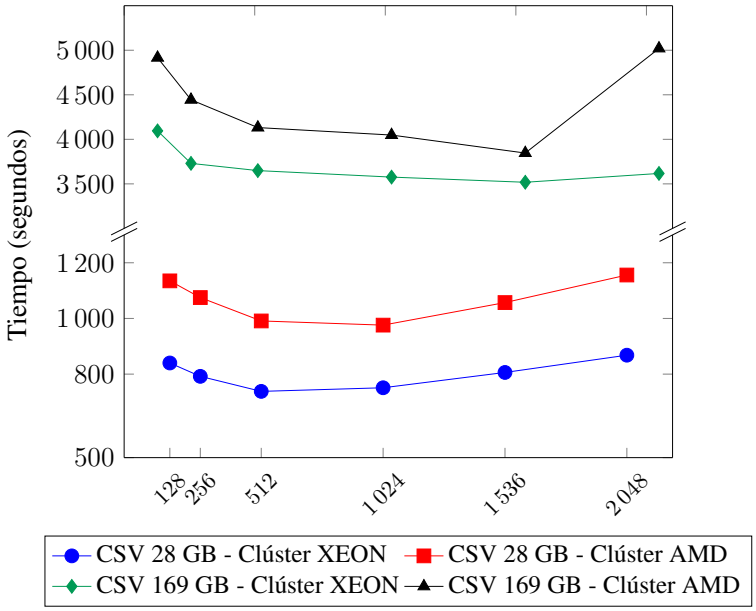


Figura 3.20: Tiempo de carga a HBase con distinto tamaños de bloque.

tiempos obtenidos en cada una de ellas. En la parte inferior se muestran los tiempos obtenidos de la carga del fichero de 28 GB en los clústers AMD y XEON, y en la parte superior el tiempos de carga para el fichero de 169 GB.

Analizando los datos obtenidos se puede apreciar que el tamaño de bloque más adecuado para cada par tamaño de fichero - clúster es aquel que permite utilizar los máximos recursos, es decir utilizar el mayor número de tareas *map* que se pueden ejecutar en paralelo, realizando el menor número de iteraciones posible.

En todos los casos cuando se incrementa el tamaño de bloque, el tiempo de carga disminuye hasta un tamaño de bloque, a partir del cual, el tiempo de carga se incremen-

ta. Por ejemplo, en la evaluación realizada en el clúster AMD con el fichero de 28 GB, el tiempo de carga diminuye desde los 1135 segundos con tamaño de bloque de 128 MB a los 976 segundos con tamaño de bloque de 1024 MB. A partir de ese tamaño, si se aumenta el tamaño de bloque el tiempo de carga también se incrementa llegando a ser de 1156 segundos, 2048 MB de tamaño de bloque. Es decir, con un tamaño de bloque de 1024 MB el tiempo se reduce en un 15 % y 16 % con respecto a los tiempo obtenidos con los tamaños de bloque de 1536 y 2048 MB.

El tiempo de cara en los XEON es considerablemente menos en los AMD, un 25 % y un 15 % menos de media en los ficheros de 28 GB y 169 GB, respectivamente.

El número de iteraciones es importante ya que implica la generación de un mayor número de tareas *map* o *reduce*. Los mejores resultados se obtienen cuando el tamaño de bloque genera un número de bloques lo más próximo a un múltiplo del número de tareas *map* ejecutadas en paralelo en un iteración, es decir, que el número de iteraciones sea el mínimo y el número de tareas sea el máximo posible. Por ejemplo, en el clúster XEON en cada iteración de tareas *map* se puede ejecutar hasta 59 tareas *map* en paralelo. Con el fichero de 28 GB para los tamaños de bloque igual o superior a 512 MB el número de iteraciones *map* es uno, ya que el número de tareas *map* no supera las 59 tareas en ninguno de los casos (56, 28, 19 y 8 tareas). Contra mayor es el tamño de bloque el número de tareas *map* es menos y cada una de ellas tiene que procesar una mayor cantidad de datos. Como las tareas *reduce* no pueden ejecutarse hasta que no terminen las tareas *map*, los recurso del servicio YARN NodeManager no va a ser asignados a ningún contenedor hasta que las tareas *map* hayan terminado. Es decir, no se están aprovechando los recursos de los YARN NodeManagers y por lo tanto se está retrasando la finalización del proceso de carga. Siguiendo con el ejemplo, para la evaluación con tamaño de bloque de 2048 MB se generan 8 bloques, es decir 8 contenedores *map* de los 59 que pueden ejecutarse en paralelo.

En el clúster AMD para el fichero de 169 GB el mejor resutlado se obtiene con tamaño de bloque de 1536 MB que da lugar a 113 tareas *map* ejecutadas en tres iteraciones con un tiempo de 3846 segundos. Un 23 % menos con respecto al peor tiempo obtenido con 2048 MB de tamaño de bloque que al igual genera tres iteraciones pero

con un menor número de tareas *map* (85 tareas). En el clúster XEON para el fichero de 28 GB el mejor tiempo obtenido (738 segundos) ha sido con el tamaño de bloque de 512 MB el cual da lugar a 56 tareas *map*, todas ellas se ejecutan en una iteración. El peor resultado se obtiene con el tamaño de bloque de 2048 MB (868 segundos), un 15 % más con respecto al mejor tiempo. Con el fichero de 169 GB el proceso de carga tarda 3518 segundos con tamaño de bloque de 1536 MB el cual genera 113 tareas *map* ejecutadas en dos iteraciones. Un 14 % más rápido que la carga realizada con tamaño de bloque 128 MB que genera 1352 tareas *map*.

Para calcular el tamaño de bloque más adecuado a cada uno de los ficheros que se van a cargar y así aprovechar al máximo el paralelismo. Una vez se conoce el número de tareas *map* que se pueden ejecutar en paralelo, el tamaño de bloque más adecuado para cada fichero se obtiene mediante la fórmula 3.4.

$$bloque\_MB = fichero\_MB/(\#iteraciones * \#max\_maps\_paralelo) \quad (3.4)$$

Donde fichero_MB es el tamaño en MB del fichero que se va a cargar, #iteraciones el número de iteraciones que se desea realizar y #max_maps_paralelo es el número de tareas *map* que se pueden ejecutar en paralelo teniendo en cuenta todas las máquinas del clúster. Por ejemplo, para el tamaño de bloque de 28 GB si el número de tareas *map* en una iteración es de 39, el tamaño de bloque adecuado es 736 MB, de esta manera todos los contenedores *map* van a procesar la misma cantidad de datos. Si el tamaño de bloque calculado es superior a 2048 MB hay que aumentar el número de iteraciones en un con respecto al número utilizado en el cálculo anterior.

3.9.3. Configuración de los recursos de YARN

En esta sección se va a comparar el tiempo de carga con las configuraciones de YARN predefinida, yarn-util.py y del inyector de carga desarrollado en esta tesis. La evaluación se ha llevado acabo en los clústers AMD y XEON con 40 HBase region servers y se han utilizado 9 ficheros diferentes. Estos ficheros corresponden a tres tablas del benchmark TPC-C [41] (Customer, Stock y Order_Line). En la tabla 3.10 se

muestran los tamaños en GB de las distintas bases de datos y de las tres tablas más grandes. En concreto para esta evaluación se van a mostrar los resultados obtenidos para las bases de datos de 1000, 3000 y 6000 warehouses.

Tabla 3.10: Tamaño de las bases de datos

Número WHs	Total (GB)	Customer (GB)		Stock (GB)		Order_Line (GB)	
		GB	#Filas	GB	#Filas	GB	#Filas
1000	**65**	13	30M	28	100M	19	300M
3000	**169**	39	90M	84	300M	62	900M
6000	**390**	77	180M	169	600M	125	1800M
12000	**781**	154	360M	337	1200M	251	3600M
18000	**1230**	231	540M	506	1800M	374	5400M
36000	**3788**	461	1080M	1012	3600M	764	10800M

Dependiendo del clúster y la configuración seleccionada se obtiene una utilización de los recursos diferente que se ve reflejada en esta evaluación. La tabla 3.8 se muestra el paralelismo obtenido para los clúters AMD y XEON. El tamaño de los bloques utilizados es diferente según la configuración YARN utilizada. Para la configuración predefinida de YARN se utiliza el tamaño de bloque predefinido al carga el fichero a HDFS (128 MB), para la configuración obtenida mediante la herramiena *yarn-utils.py* se utiliza un tamaño de bloque que genere el mínimo número de iteraciones con el máximo número de tareas map dentro de los valores 128, 256, 512, 1024, 1536 y 2048 MB. Y para la configuración de YARN obtenida para el inyector de carga se ha utilizado la formula 3.4 (ver anexo A).

Las figuras 3.21, 3.22 y 3.23 muestran los tiempos de carga para las bases de datos de 1000, 3000 y 6000 warehouses en los clusters AMD y XEON utilizando las diferentes configuraciones de YARN. En cada una de ellas las barras azules son las que representan la configuración predefinida de YARN, las rojas se corresponden a la configuración obtenida mediante la herramienta yarn-util.py y las verdes la configuración para el inyector de carga. Las barra con color son los resultados de las ejecuciones en el clúster AMD, mientras que las barras sin relleno pertenecen al clúster XEON. Cada una de las evaluaciones se ha ejecutado tres veces y se presenta la media tiempos obtenidos en las tres ejecuciones.

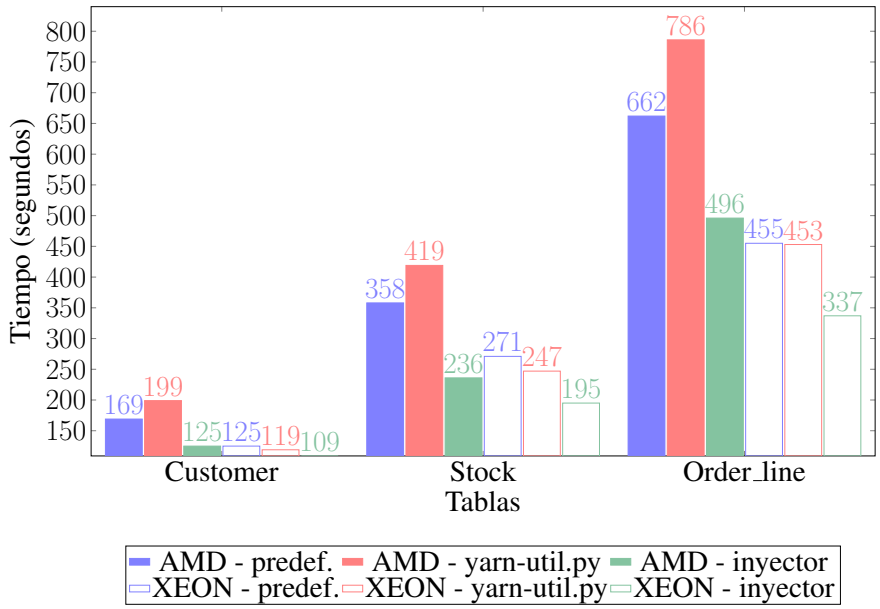


Figura 3.21: Tiempo de carga en HBase. 1000 warehouses

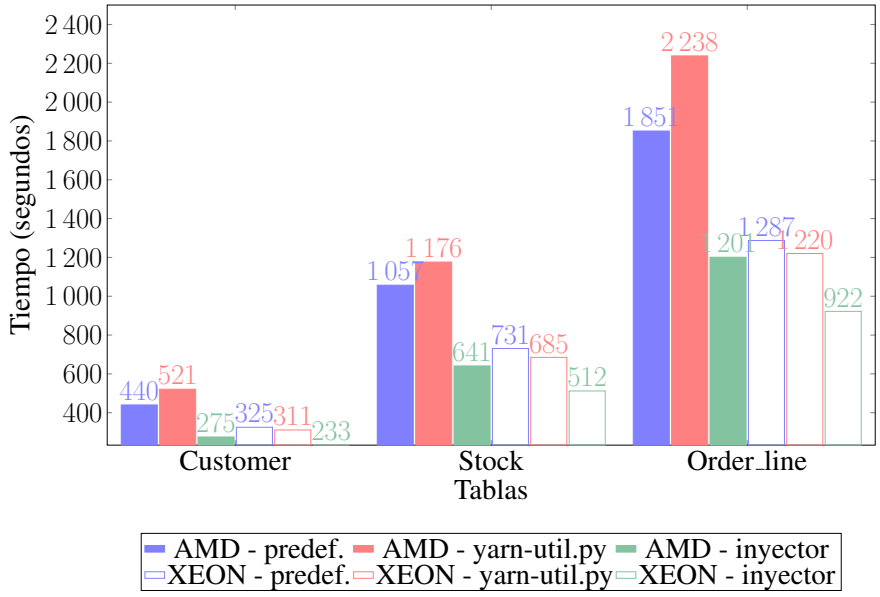


Figura 3.22: Tiempo de carga en HBase. 3000 warehouses

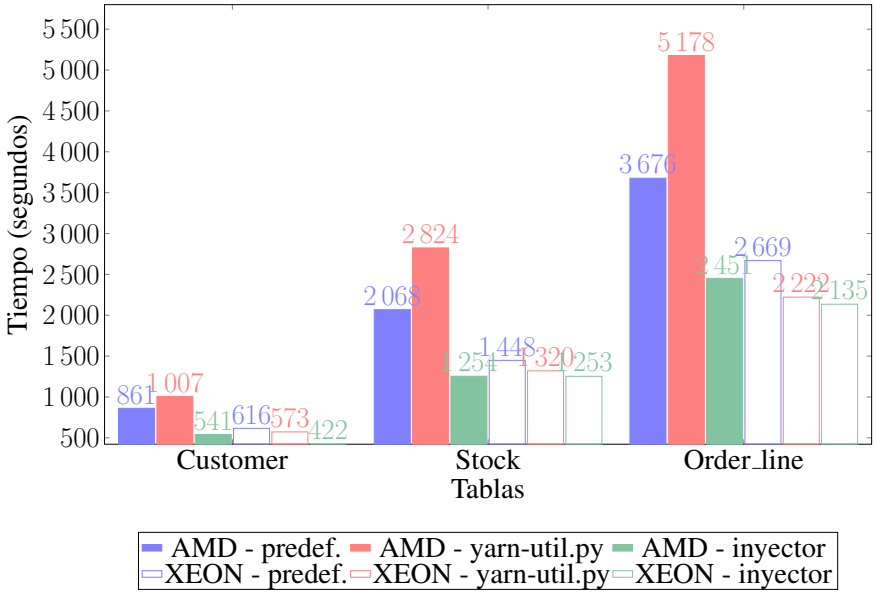


Figura 3.23: Tiempo de carga en HBase. 6000 warehouses

Los peores resultado se obtienen con la configuración obtenida mediante la herramienta *yarn-util.py* en el clúster AMD (199 segundos la tabla Customer, 419 segundos tabla Stock y 786 tabla Order_Line, base de datos de 1000 warehouses, figura 3.21). Mientras que los mejores resultados se obtienen con la configuración presentada en esta tesis, inyector de carga (125 segundos la tabla Customer, 236 segundos tabla Stock y 496 tabla Order_Line, base de datos de 1000 warehouses, figura 3.21). En concreto los tiempos mejoran el 38 % y 28 % en comparación con la configuración *yarn-util.py* y la configuración predeterminada con la carga de la base de datos de 1000 warehouses. El 46 % y 37 % con la base de datos de 3000 warehouses y 51 % y 36 % con la base de datos de 6000 warehouses y las configuraciones *yarn-util.py* y predeterminada, respectivamente. Por ejemplo, al cargar la tabla Order_Line de la base de datos de 6000 warehouses (figura 3.23) y con la configuración *yarn-util.py* el tiempo que tarda en finalizar es 5178 segundos, 3676 segundos con la configuración predeterminada y el mejor resultado se ha obtenido con la configuración del inyector de carga, 2451 segundos. Esto se debe a que el paralelismo obtenido con la configuración del inyector de carga (119 tareas *map* y 59 tareas *reduce*) es superior al obtenido con la configuración

predefinida (78 tareas *map* y *reduce*) y *yarn-util.py* (39 tareas *map* y *reduce*). Además al tener 40 regiones cada cada una de las tablas, con la configuración obtenida mediante la herramienta *yarn-util.py*, durante el proceso *reduce* realiza dos iteraciones, la primera con 39 tareas *reduce* y la segunda con una sola tareas *reduce* aumentando el tiempo de carga de las tablas.

Los mejores resultados en el clúster XEON vuelven a ser, al igual que en el clúster AMD, los obtenidos mediante la configuración del inyector de carga (109 segundos la tabla Customer, 195 segundos tabla Stock y 337 tabla Order_Line, base de datos de 1000 warehouses, figura 3.21)), debido a que se obtene un mayor paralelismo en comparación con el resto de configuraciones (99 tareas *map* y 49 tareas *reduce*). En concreto los resultados mejoran en la base de datos de 1000 warehouses el 22 % y 18 % de media con respecto a los resultados obtenidos con las configuraciones predeterminada y *yarn-util.py*. Lo mismo ocurren con el resto de bases de datos donde el tiempo de carga mejora un 42 % y 25 % de media con la base de datos de 3000 warehouses y 12 % y 21 % de media con la de 6000 warehouses con las configuraciones predeterminada y *yarn-util.py*, respectivamente. La configuración que ha dado peores resultado en esta evaluación es la predeterminada con un paralelismo de (78 tareas *map* y *reduce*), el cual al pesar de tener mayor paralelismo que el obtenido mediante la herramienta *yarn-util.py* (59 tareas *map* y *reduce*), el número de bloques que se generan al cargar el fichero en HDFS con tamaño de bloque 128 MB hace que se necesite un número muy alto de iteraciones (8, 17 y 13 iteraciones para las tablas Customer, Stock y Order_Line con la base de datos de 6000 warehouses). Por otro lado, en esta ocasión el paralelismo obtenido en el número de tareas reduce con la configuración *yarn-util.py* permite realizar todas las tareas reduce (40) en una sola iteración reduciendo el tiempo de carga. Por ejemplo, la tabla Stock de la base de datos de 3000 warehouses (figura 3.22)) el tiempo obtenido mediante la configuración del inyector de carga es de 233, 512 y 922 segundos para las tablas Customer, Stock y Order_Line, respectivamente. Mientras que para la configuración *yarn-util.py* el tiempo de carga es de 311, 685 y 1220 segundos y con la configuración predefinida el tiempo de carga es de 325, 731 y 1287 segundos para las tablas Customer, Stock y Order_Line, respectivamente.

3.9.4. Escalabilidad del Inyector de carga

Configurar todos los servicios de manera que se aproveche al máximo todos los recursos de cada una de las máquinas mediante la asignación de afinidad a los distintos servicios y configurando YARN de tal manera que se consiga el máximo paralelismo posible permite demostrar la escalibilidad en tres escenarios diferentes con bases de datos que van desde los 65 GB (1000 warehouses) a los 3,7 TB (36000 warehouses). Los tamaños de bloque utilizados en cada una de las evaluaciones muestran en el anexo B, cada una de las ejecuciones ha sido realizad 3 veces y se muestran los tiempos medios obtenidos para cada una de ellas.

Las figuras 3.24, 3.25 y 3.26 muestran los tiempos de carga en los clústers AMD, XEON y Bullion para las tablas Customer, Stock y Order_Line. En la figura 3.24 se muestran los tiempos en el clúster AMD para las bases de datos de 1000, 3000, 6000, 12000 y 18000 warehouses (WHs). Por ejemplo para la tabla Stock, el tiempo de carga con las bases de datos de 1000, 3000, 6000 WHs es 236 segundos, 641 y 1254 segundos respectivamente. En las tres ejecuciones el número de iteraciones *map* es uno y el número de tareas *map* a ejecutar también es el mismo, 119. Por lo que a pesar de que la cantidad de datos a procesar se triplica y sixtuplica, el tiempo de carga prácticamente se incrementa tres y seis veces. Si se comparan los tiempos para 6000, 12000 y 18000 warehouses, donde el tamaño de bloque utilizado para subir el fichero a HDFS es el mismo (1450 MB), con 6000 warehouses el número de tareas *map* es 119, con 12000 warehouses sea 238 y 357 tareas *map* con 18000 warehouses. Por lo que el número de iteraciones *map* a ejecutar van a ser 1, 2 y 3 respectivamente. Este hecho se refleja en los tiempos de carga donde para la tabla Stock de la base de datos de 6000 WHs el tiempo es de 1254 segundos, para la de 12000 WHs es de 2499 segundos y para la de 18000 WHs es de 4227 segundos.

Exactamente el mismo comportamiento se puede observar con el resto de tablas y desplieges. Analizando los tiempos obtenidos del Bullion (figura 3.26, escala de manera lineal hasta los 3,7 TB que se han evaluado. Por ejemplo, analizando los tiempos de la tabla Order_Line, para los tamaños de datos 1000, 3000 y 6000 WHs todas las

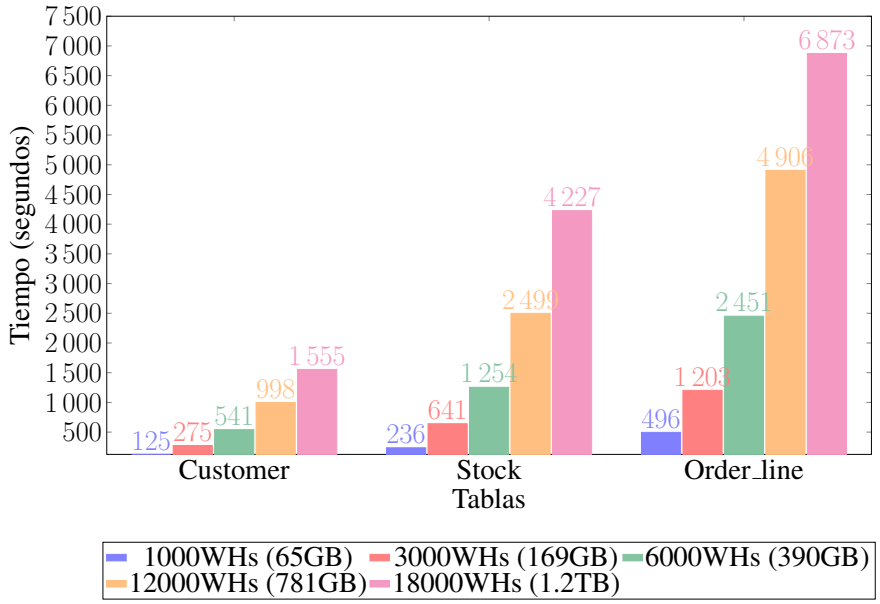


Figura 3.24: Tiempo de carga en HBase en el clúster AMD

tareas *map* se ejecutan en un única iteración donde se puede ver que los tiempos casi llegan a triplicarse y duplicarse despues, 1047 segundos con el tamaño de 1000 WHs, 2253 con el de 3000 WHs y 4447 segundos con el tamaño de 6000WHs. Con el resto de tamaños se puede observar que el tiempo se triplica desde la base de datos de 6000 WHs a la de 18000 WHs donde se tarda 13446 segundos en realizar la carga de la tabla Order_Line, además el número de iteraciones *map* pasa de ser 1 a 3. Desde la misma tabla con tamaño de la base de datos de 6000 WHs se puede ver que el tiempo se aumenta seis veces con la carga de la tabla de la base de datos de 36000 WHs, con un tiempo de 22499 segundos y el número de iteraciones *map* también pasa a ser 6.

3.9.5. Mejora en el rendimiento de carga

En esta última evaluación se utilizan todas las contribuciones realizadas en este capítulo. Para ello se ha realizado la carga de un fichero de 13 GB que contiene 90 millones de filas utilizando distintos procesos para almacenar los datos en una tabla en HBase en el clúster AMD. La figura 3.27 muestra los tiempos de carga de 5 técnicas diferentes: 1) HBase Put: Es la manera más básica de insertar datos en HBase,

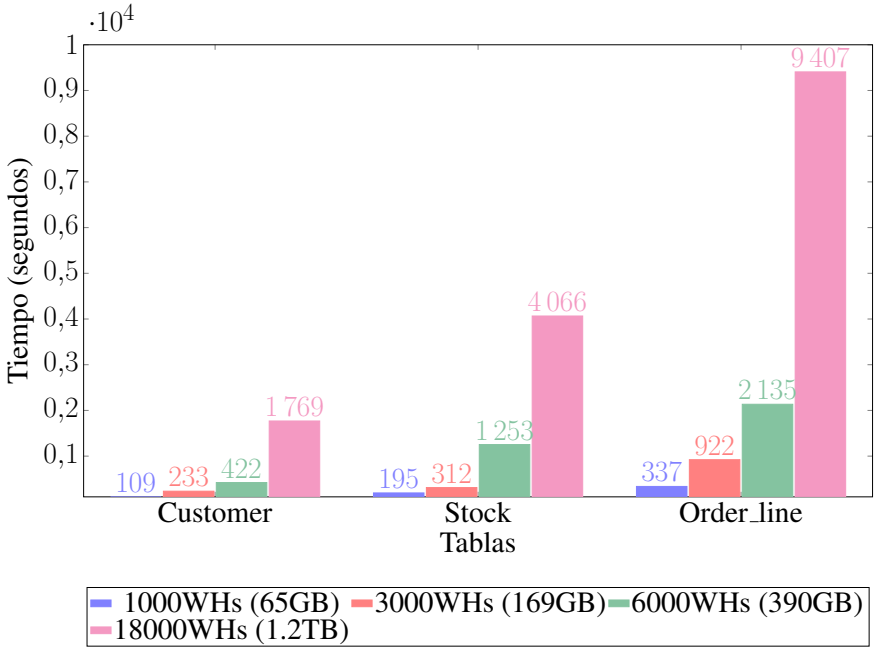


Figura 3.25: Tiempo de carga en HBase en el clúster XEON

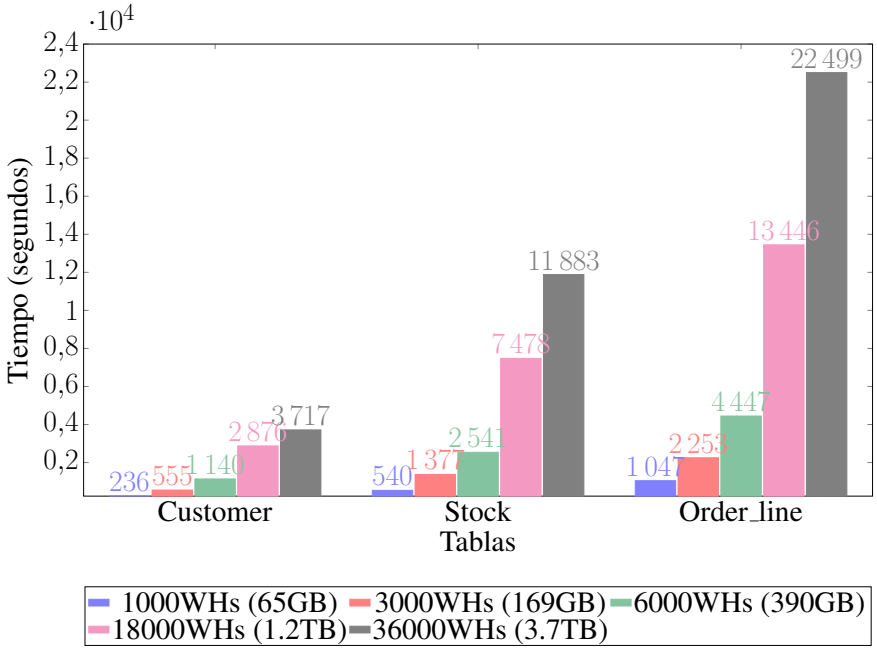


Figura 3.26: Tiempo de carga en HBase en el Bullion

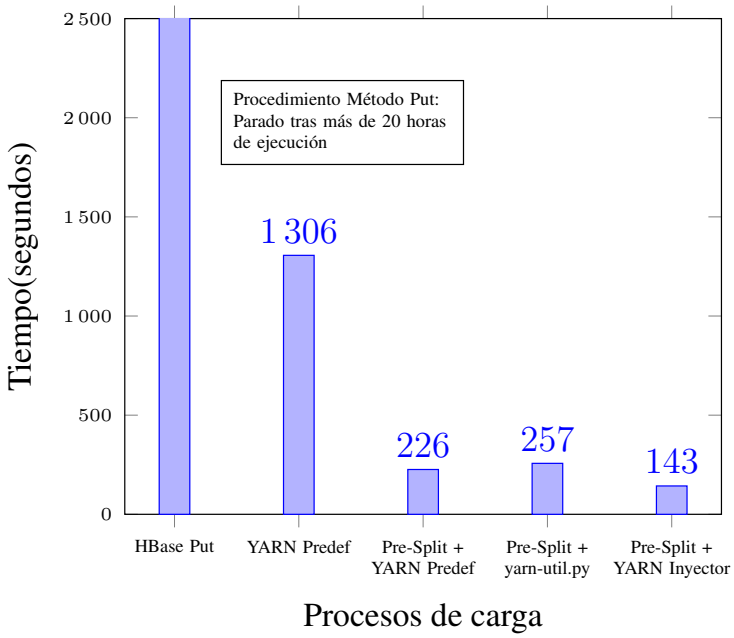


Figura 3.27: Tiempo de carga (s) con diferentes procedimientos

mediante el método *Put* disponible en la API Java de HBase, presentado en la Sección 3.2. 2) YARN Predef: Se realiza la carga de los datos utilizando el inyector de carga presentado en la Sección 3.4 utilizando la configuración predefinida de YARN (Sección 3.3.0.1), en esta evaluación la tabla consta de una región al inicio de la carga. 3) Pre-Split + YARN Predef: Al igual que en la evaluación anterior se utiliza el inyector de carga con la configuración predefinida de YARN pero en esta ocasión la tabla ha sido dividida en 40 regiones antes del comienzo de la carga, tantas como HBase region servers en el clúster. 4) Pre-Split + yarn-util.py: En esta ocasión la tabla ha sido dividida en 40 regiones y YARN ha sido configurado con la salida obtenida de la herramienta yarn-util.py (Sección 3.3.0.2). Y finalmente, 5) Pre-Split + Inyector: Al igual que en las dos últimas evaluación la tabla es dividida en 40 regiones antes de comenzar el proceso de carga, pero en esta ocasión YARN es configurado para hacer más eficiente el Inyector de carga, tal y como se muestra en la Sección 3.7.0.1, además de distribuir lo recursos de las máquinas para obtener el máximo rendimiento de los mismos (Sección 3.6.

La evaluación realizada con el método HBase Put ha sido detenida manualmente tras 22 horas de ejecución, durante este tiempo se han creado 3 regiones de manera automática mediante los mecanismos de creación de regiones de HBase (Sección 3.2.1). Todas ellas permanecen en el mismo HBase region server y únicamente se han cargado 25 millones de filas de los 90 millones que contiene el fichero, es decir, un 22,5 % de los datos han sido cargados tras casi un día de ejecución. Realizando la misma carga utilizando el Inyector de carga con la configuración predefinida de YARN (YARN Predef) y cargando previamente el fichero en HDFS utilizando el tamaño de bloque predefinido (128 MB) se ha llegado a cargar por completo en 1306 segundos, tal y como se muestra en la segunda columna de la gráfica. Una vez se termina el proceso de carga todos los datos permanecen en la misma máquina de clúster y el proceso *reduce* no ha sido paralelizado ya que solo hay una región de la tabla.

La evaluación realizada con la configuración predefinida de YARN pero creando varias regiones antes del comienzo en el proceso de carga el tiempo total de carga es de 227 segundos, tal y cómo se muestra en la tercera columna de la figura 3.27 (Pre-Split + YARN Predef). Es decir se consigue una mejora en el tiempo de carga de un 83 % al incorporar la generación de regiones mediante el método pre-split (sección 3.5), una de las aportaciones de este capítulo al proceso de carga. Utilizando esta aportación que permite crear regiones equilibradas antes del comienzo del proceso de carga y esta vez configurado YARN con la salida obtenida de la herramienta de HortonWorks yarn-util.py, el tiempo de carga obtenido es 257 segundos. Este resultado se debe a que la configuración obtenida no es la adecuada para este trabajo MapReduce, lo supone que no se aprovechen de manera adecuada los recursos asignados a los procesos MapReduce haciendo que el tiempo obtenido sea un 12 % superior al tiempo obtenido con la configuración predefinida de YARN.

Finalmente se han utilizado todas las aportaciones de este capítulo para realizar la evaluación donde se ha creado las regiones equilibradas antes del proceso de carga mediante el Pre-Split (sección 3.5), se han distribuidos los recursos de las máquinas del clúster y se han asociado los servicios a distintos nodos NUMA (sección 3.6), y finalmente se ha configurado YARN utilizando la configuración más adecuada para el

Inyector de carga (sección 3.7). Tras la ejecución el tiempo de carga obtenido es de 143 segundos lo que supone un 89 %, 38 % y un 45 % más rápido en comparación con los tiempos obtenidos mediante la configuración predefinida de YARN sin dividir la tabla antes de la carga y creando las regiones, y mediante la configuración obtenida con la herramienta yarn-util.py.

3.10. Conclusiones

A lo largo de este capítulo se han presentado varias aportaciones al proceso de carga de datos desde ficheros a bases de datos NoSQL como es HBase. Como resumen de todas las aportaciones se ha realizado una demostración de como el proceso de carga se ve mejorado gracias al uso de las diferente aportaciones. La aportaciones son: 1) Implementación de un método de carga basado en la herramienta ImportTSV que permite generar las claves de las filas inyectadas durante el proceso de carga sin necesidad de procesar el fichero con los datos antes de comenzar la carga, sección 3.4. 2) Una herramienta que permite crear regiones las cuales van a tener aproximadamente la misma cantidad de datos cuando la tabla está vacía mediante el muestreo del fichero de datos, sección 3.5. 3) Implementación de un sistema que permite distribuir los recursos de cualquier máquina con arquitectura NUMA entre una serie de servicios. Este sistema se puede utilizar independientemente de los servicios que se quieran desplegar, sección 3.6. 4) Implementación de una herramienta que permite configurar YARN distribuyendo los recursos que le han sido asignados con la herramienta anterior de tal manera que se consiga el mayor paralelismo posible a la hora de realizar la carga de datos, sección 3.7.

Este trabajo fue presentado en los congresos 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing con el título Massive Data Load on Distributed Database Systems over HBase [42] y 8th International Conference on Data Science, Technology and Applications (DATA 2019) con el título Parallel Efficient Data Loading [43].

Capítulo 4

Equilibrado de Datos Uniforme

HBase, como otras tecnologías NoSQL y bases de datos distribuidas, usan varios nodos para almacenar grandes cantidades de información que no pueden ser almacenados en un solo nodo. Además explotan el sistema distribuido al realizar consultas que son ejecutadas en paralelo en las distintas máquinas. Para conseguir que este proceso realmente sea en paralelo y se utilice el máximo número de recursos, las consultas tienen que ser ejecutadas entre las distintas máquinas. Para ello los datos a los que acceden tienen que estar distribuidos de manera uniforme.

Si se conoce la aplicación, las consultas de la misma y los datos no cambian, bastará con una distribución inicial uniforme de los datos en el momento de la carga. Si se añaden datos o se añaden o eliminan máquinas del clúster, la configuración inicial puede dejar de ser eficiente porque hay regiones con más datos que otras y/o porque hay máquinas que no tienen datos (no son accedidos por las consultas).

Cuando las regiones tienen el mismo número de claves el tiempo de respuesta (latencia) al realizar consultas que acceden de manera uniforme a los datos es el mismo en todas ellas. Para asegurar la misma latencia en todas las regiones en los tres escenarios mostrados en la figura 4.1, hay que equilibrar los datos de la tabla entre todas las máquinas del clúster. HBase proporciona políticas de balanceo de carga que controlan la carga de las regiones y los nodos del sistema, sección 4.1.

En las siguientes secciones se presenta un algoritmo que permite distribuir de manera uniforme los datos en regiones y máquinas del clúster (sección 4.2), principal

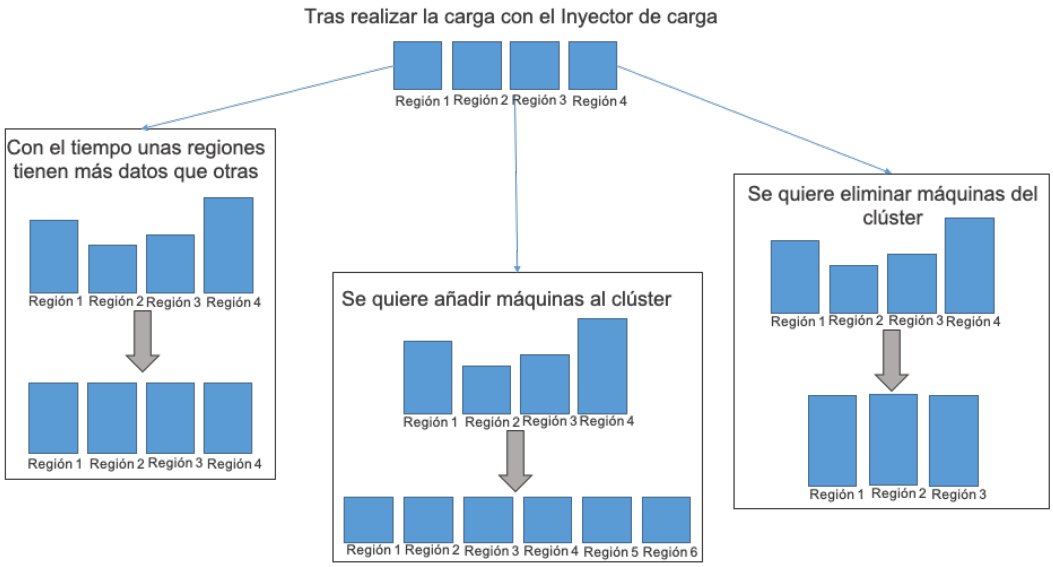


Figura 4.1: Escenarios en los que las regiones pueden no estar equilibradas con respecto al clúster

aportación de este capítulo. Finalmente, se va a presentar una evaluación del algoritmo en un clúster con nodos homogéneos donde se va a mostrar la distribución inicial de los datos, los tiempos de ejecución de cada una de las fases del algoritmo de equilibrado y la distribución final de los datos en la seccion 4.3 y para terminar se presentarán las conclusiones en la sección 4.4.

4.1. Mecanismos de balanceo de carga en HBase

El rendimiento de HBase puede verse gravemente afectado por una distribución de las regiones no adecuada haciendo que unos HBase region servers estén sirviendo más regiones y datos que otros. Esta distribución no adecuada de los datos puede hacer que alguna de las máquinas no tengan recursos suficientes de CPU y memoria para poder gestionar la cantidad de datos que tiene mientras que otras máquinas se encuentren con muy poca carga, ya que apenas están sirviendo datos. Para solucionar esta situación, HBase lanza cada 5 minutos el balanceador de carga (sección 2.5.3.2).

Todas las políticas de balanceo proporcionadas por HBase gestionan la ubicación

de las regiones entre los nodos del sistema pero no diferencian las regiones teniendo en cuenta la tabla a la que pertenecen. En todas ellas, las regiones son tratadas por igual y no se realiza ningún cambio sobre las mismas, es decir si una región está sirviendo muchos datos en comparación con otra, las regiones siguen estando igual. El principal objetivo de las políticas de balanceo es asegurar que el consumo de recursos de los nodos del sistema es igual por lo que puede llegar a darse que un HBase region server tenga una región y otro HBase region server tenga 20 regiones, por ejemplo.

4.2. Regiones con cantidades de datos uniforme

El equilibrado de datos uniforme pretende asegurar que la cantidad de claves servida por cada una de las regiones de una tabla es la misma en cada uno de los HBase region servers. De esta manera la latencia de las consulta que acceden de manera uniforme a los HBase region servers, es la misma en todos ellos permitiendo obtener el resultado en el mínimo tiempo posible.

Para poder distribuir equitativamente los datos se necesita conocer la distribución de los mismos y el número de instancias de HBase region server activas, así se puede determinar si la distribución actual de los datos está equilibrada o por el contrario hay regiones que están sirviendo una mayor cantidad de datos que otras. En esta sección se presentan dos contribuciones, el histograma (sección 4.2.1), el cual permite conocer la distribución de las claves en las distintas regiones de la tabla y el algoritmo de equilibrado de datos uniforme de las regiones (sección 4.2.2).

4.2.1. Histograma

El histograma tiene un papel fundamental en el proceso de crear regiones equilibradas ya que da a conocer la distribución inicial de los datos. Además de mostrar la distribución de las claves en las regiones de la tabla, durante la construcción del histograma se obtiene el número de tuplas de cada una de las regiones, el número de tuplas que tiene la tabla en total, el número de tuplas que debería de tener cada región para que todas tengan la misma cantidad de tuplas y la cantidad de tuplas que se encuentran

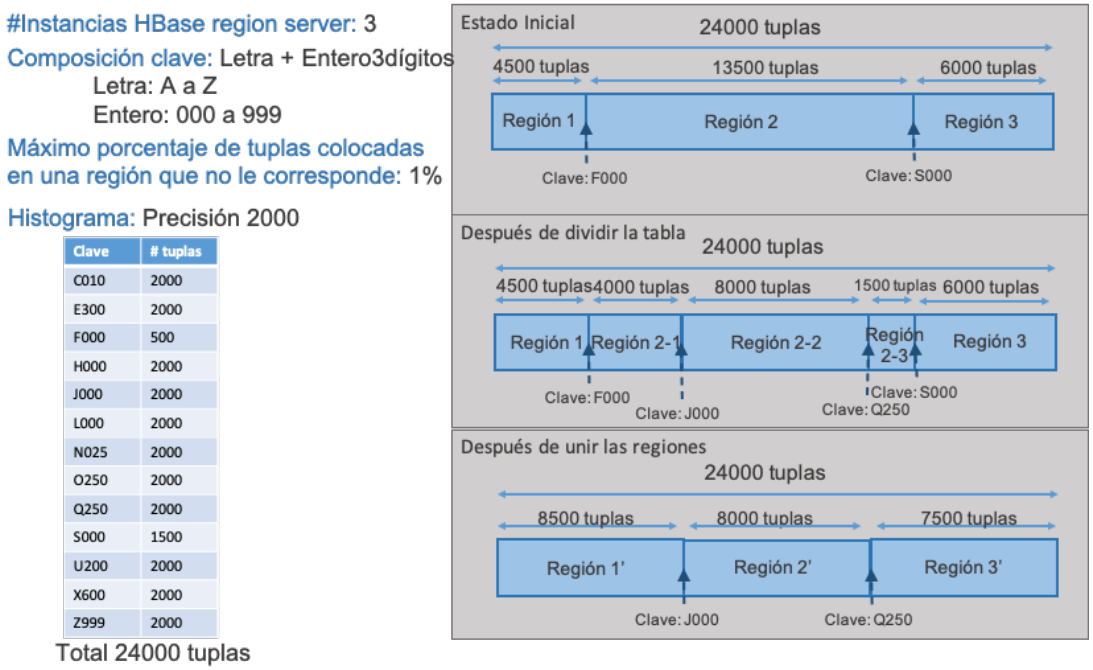


Figura 4.2: Ejemplo de histograma y principales pasos para la creación de las regiones equilibradas

en una región que no le corresponde. Por ejemplo, en el escenario presentado en la figura 4.2 donde se muestra una tabla dividida en tres regiones en la que las claves están compuestas por una letra que va desde la "A" a la "Z" seguido de un número de tres cifras que va desde el valor 000 al 999, las cuales tienen 4500, 13500 y 6000 tuplas respectivamente haciendo un total de 24000 tuplas (Estado inicial).

Para crear el histograma se realiza un proceso que recorre cada una de las regiones de la tabla en paralelo y obtiene las claves de las tuplas que se encuentran cada X tuplas, este número se define como precisión y permite ajustar el tamaño del histograma. En el ejemplo presentado en la figura 4.2 la precisión del histograma es de 2000 tuplas, de la Región 1 se han obtenido las claves de las tuplas en la posición 2000, 4000 y 4500, ya que esta región tiene 4500 tuplas. Para la Región 2 las claves obtenidas has sido en la posición de las tuplas 2000, 4000, 6000, y así hasta la 12000 y 13500, con un total de 13500 tuplas. Finalmente la Región 3 con 6000 tuplas, las claves obtenidas son las de las tuplas que se encuentran en las posiciones 2000, 4000 y 6000. Cuando la

cantidad de tuplas de una región no coincide con un múltiplo de la precisión, la última clave del histograma para dicha región va a indicar las tuplas que quedan restantes en la región y la clave de la última fila, tal y como se puede ver que ocurre en las regiones Región 1 y Región 2.

El valor de la precisión del histograma es indicado por el usuario y por omisión el valor utilizado es 10000. Este valor tiene un impacto tanto en la cantidad de información que se va a obtener del histograma como en el tiempo de ejecución. Si el tamaño de precisión es muy pequeño la cantidad de información obtenida va a ser mayor y va a permitir obtener las claves por las que generar las nuevas regiones de manera más precisa, pero el tiempo que lleva generar el histograma es mayor ya que tiene que acceder a un mayor número de claves y tiene que enviar más información al cliente. Si el tamaño de la precisión es muy grande, puede darse el caso que el tamaño de las regiones sea inferior a la precisión lo cual no va a permitir conocer la distribución de las claves dentro de la región y la generación de las nuevas regiones no se va a ajustar al tamaño deseado.

El histograma se ha codificado como un *coprocessor* de HBase, es decir es un proceso que se ejecuta en cada uno de los HBase region servers. Cada uno de los *coprocessors* va a recorrer las claves de la región de la cual va a obtener el histograma y cada X claves, siendo X la precisión, va a generar una salida indicando la clave de dicha posición y la cantidad de claves que ha recorrido hasta llegar a dicha posición. La salida es enviada al cliente, donde obtendrán los resultados obtenidos de cada uno de los *coprocessors* reduciendo de esta manera el tráfico de red y paralelizando la generación de histograma. Si el histograma se crease como una consulta desde el lado del cliente, se enviarían todas las claves de cada una de las regiones al cliente y luego se recorrerían todas ellas de manera secuencial para generar el histograma. En este caso tanto el trafico de red como el tiempo de ejecución serías mucho mayor.

4.2.2. Equilibrado de Datos Uniforme

El equilibrado de datos uniforme solo se va a llevar acabo si el tamaño de algunas regiones ha crecido mucho más que las otras o si el número de HBase region servers ha cambiado con el tiempo.

El primer paso para obtener las regiones equilibradas es obtener los punto de división, es decir las claves que delimitan las regiones. Dependiendo de la cantidad de cada región de la tabla y de la precisión del histograma estos puntos se obtienen de dos maneras diferentes. La primera, cuando el tamaño de las regiones y la precisión del histograma no permiten obtener una representación detallada de la distribución de las claves, en cuyo caso se recorre la tabla de manera secuencial para encontrar los puntos de división. La segunda, cuando el tamaño de la tabla es suficientemente grande como para obtener un histograma que permita obtener los puntos de división de manera precisa. En este caso se recorre el histograma para obtener lo puntos de división.

En el ejemplo de la figura 4.2 con la información obtenida del histograma y la información que aporta HBase sobre el número de HBase region servers (3) y el número de regiones de la tabla (3), el equilibrado de datos uniforme se va a lanzar debido a la diferencia de tamaño entre las regiones de la tabla ya que el número de HBase region servers y el número de regiones de la tabla es el mismo. El número total de tuplas en la tabla es 24000, por lo que para crear regiones con el mismo número de claves cada una de ellas tiene que tener 8000. La Región 1 tiene 4500 claves (3500 claves menos de las esperadas), la Región 2 tiene 13500 claves (5500 claves de más) y la región 3 tiene 6000 (2000 menos), en total hay 5500 tuplas que están siendo servidas por otra región (22,9 %). Para saber si hay que ejecutar el equilibrado de carga uniforme, se define un parámetro que indica el porcentaje máximo de tuplas que pueden estar en una región distinta a la esperada, en este caso se ha establecido a 1 %.

Para obtener los puntos de división de la tabla del ejemplo, se calcula si el tamaño de la tabla con respecto a la precisión del histograma va a permitir tener una representación lo suficientemente buena de la distribución de la claves. Para ello se calcula si el número total de filas es mayor al número de filas del histograma en toda la tabla,

teniendo en cuenta que al menos la tabla tiene tanta tuplas como dos veces el tamaño de la precisión. En el ejemplo de la figura 4.2 el histograma tiene la suficiente precisión, para obtener los punto de dividisión se recorre el histograma y se obtiene la clave que se encuentra en la posición de la tabla de tal manera que la suma de las anteriores entradas sea igual o superior al tamaño deseado. Por ejemplo en el histograma de la figura 4.2, la clave "J000" será el primer punto de split con 8500 tuplas en total y el siguiente punto de split será "Q250" con 8000 tuplas, de esta manera la última región tendrá un total de 7500 tuplas.

Una vez calculados los puntos de división, se crean las regiones utilizando el método *split* (sección 3.5.1). Este método permite crear dos nuevas regiones de una inicial indicando un punto de división. Las nuevas regiones creadas permanecen en el mismo HBase region server en el que se encontraba la región inicial y por tanto los datos siguen almacenados en la misma carpeta de HDFS en dos HFiles distintos, uno por región. Siguendo con el ejemplo de la figura 4.2, los puntos de división se corresponden con claves que se encuentran en la Región 2, al crear la nuevas regiones esta región queda dividida en tres nuevas regiones (Región 2-1, Región 2-2 y Región 2-3)

El siguiente paso consiste en mover las regiones desde el HBase region server en el que se encuentra al hacer el *split* al HBase region server donde se encuentra la región a la que queremos unirla. Este paso es necesario para mover los datos desde la carpeta HDFS en la que se encuentran los datos del HBase region server inicial a la del HBase region server que va a servir los datos de la nueva región. Por ejemplo, la Región 1 y Región 2-1 van a ser unidas para que la región resultante Región 1' tenga los datos de ambas. Para mover las regiones de un HBase region server a otro se utiliza el método *move* de HBase, este método necesita el identificador de la región y el identificador del HBase region server al que se va a mover la región. La parte a) de la figura 4.3 muestra las regiones creadas tras realizar las divisiones, las nuevas regiones Región 2-1, Región 2-2 y Región 2-3 se encuentran en el HBase region server 2, las regiones Región 2-1 y Región 2-3 tiene que ser movidas a las instancias HBase region server 1 y HBase region server 3 respectivamente para que cada HBase region server esté

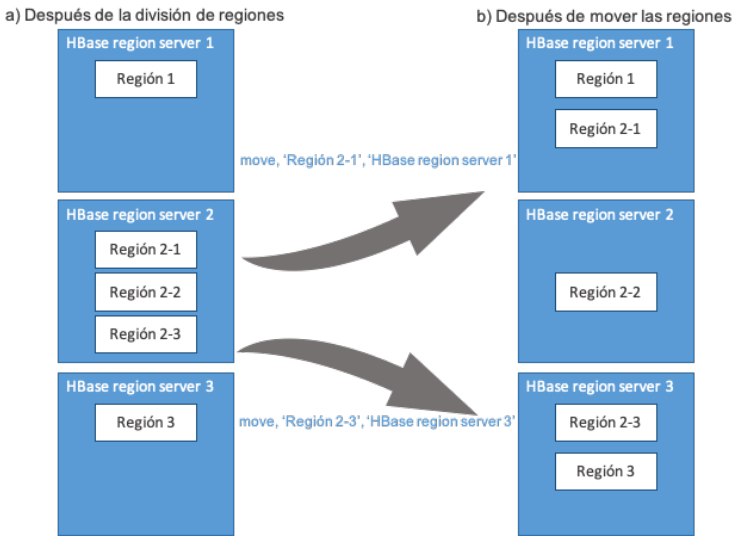


Figura 4.3: Ejemplo método *move* de HBase de dos regiones

sirviendo la misma cantidad de datos. Tras terminar el proceso *move* las regiones están en sus respectivos HBase region servers como se muestra en la parte b) de la figura 4.3.

En este punto, cada HBase region server tiene los datos que le corresponde servir pero sigue diferenciando las regiones. Este estado puede repercutir en el rendimiento de las consultas ya que en caso de que tenga que acceder a todos los datos, HBase tendría que leer de los dos ficheros lo que supone un tiempo extra al abrir y cerrar los ficheros si los datos no los tiene en memoria. Además las lecturas en distintas regiones se realizan de manera secuencial por lo que no se puede paralelizar la lectura de regiones dentro del mismo HBase region server con peticiones realizadas desde el cliente. Por ese motivo se ha decidido agrupar las regiones, en caso de que el HBase region server no tenga suficiente memoria como para almacenar los datos de todas las regiones de todas las tablas que esté sirviendo, reducir el número de regiones al mínimo va a evitar que se estén abriendo y cerrando ficheros continuamente.

En la parte a) de la figura 4.4 se muestran las regiones en sus respectivos HBase region servers, la instancia HBase region server 1 está sirviendo las regiones Región 1 y Región 2-1, la Región 2-2 se encuentra en el HBase region server 2 y las regiones Región 2-3 y Región 3 se encuentran en la instancia HBase region server 3. Para unificar

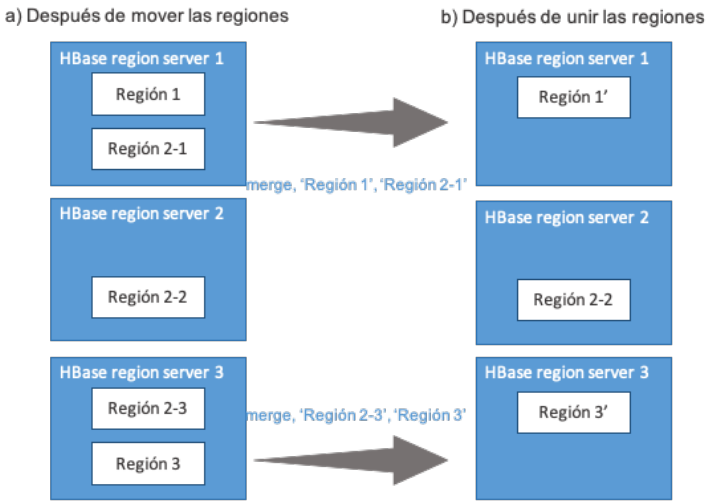


Figura 4.4: Ejemplo método *merge* de HBase

las regiones, se utiliza el método *merge* de HBase el cual permite unir dos regiones de la misma tabla, con claves consecutivas y que se encuentran en el mismo HBase region server en una misma región de manera lógica. Es decir, las regiones serán unificadas para HBase en una misma región pero los datos permanecerán en dos HFiles distintos (en el siguiente paso se unifican los HFiles). Para utilizar el método *merge* hay que indicar los identificadores de las dos regiones a unir en el orden en el que se van a unificar, en caso de que las regiones no posean claves consecutivas o sean de distintas tablas el proceso *merge* fallará. En la figura 4.4 se unifican las regiones Región 1 y Región 2-1 dando lugar a la región Región 1' y las regiones Región 2-3 y Región 3 creando la nueva región Región 3'.

Finalmente para unificar los HFiles, HBase posee un método llamado *major compact*. No es necesario aplicar este método pero si no se hace, si se realiza una consulta sobre la región en la que se recorran todas las tuplas y si los datos no están en memoria, cuando termine de leer uno de los HFiles tendrá que abrir y comenzar al leer del siguiente aumentando el tiempo de respuesta. Además si la tabla tiene varias familias de columnas, va a haber tantos HFiles como familias de columnas y regiones aumentado de esta manera el número de veces que el HBase region server tiene que acceder a los datos. En la figura 4.5 se muestra en la parte superior el estado en el que se encuentra

Figura 4.5: Ejemplo método *major compact* de HBase

las regiones y los HFiles después de haber realizado el *merge* y en la parte inferior tras haber realizado el proceso *major compact*. Para ejecutar el *major compact* hay que indicar el identificador de la región que se quiere compactar y los HFiles de dicha región se unificaran dando lugar a tantos HFiles como familias de columnas tenga la tabla.

El algortimo 5 muestra los distintos pasos del proceso de equilibrado de datos uniforme. Como entrada recibe tres valores 1) el nombre de la tabla, 2) la precisión del histograma y 3) el máximo porcentaje de tuplas que se encuentran en la región que no le corresponde. El primer paso a ejecutar es la generación del histograma para poder obtener la distribución de las claves y obtener el número de tuplas de la tabla, el número de tuplas de cada región, el número de regiones, el número de tuplas que debería de tener cada región para que cada HBase region server sirva la misma cantidad de datos y el porcentaje de tuplas que se encuentran en la región que no le corresponde. Además de obtener de HBase el número de instancias de HBase region server activas mediante el método *getCurrent#HRS*, (líneas 1 a 6, algoritmo 5).

Una vez se ha obtenido toda la información necesaria se comprueba si se puede lanzar el equilibrado de datos uniforme comprobando si el número de instancias HBase region servers es diferente al número de regiones de la tabla o si el porcentaje de

tuplas colocadas en una región que no le corresponde es mayor al máximo porcentaje indicado como parámetro del algoritmo (línea 7, algoritmo 5). En caso de que se ejecute el equilibrado de datos uniforme: 1) se obtención de los puntos de división (línea 8). 2) Creación de las regiones con los puntos de *split* (línea 9). 3) Mover las regiones desde la instancia HBase region server en la que se encuentra tras el *split* hasta la instancia donde se encuentra la región con la que se va a unir y, finalmente, unirlas de manera lógica para que la instancia HBase region server vea una única región (línea 10). 4) Unificar los HFiles con los datos de la región para que los datos de cada familia de columnas se encuentren en el mismo fichero en HDFS (línea 11). 5) Se almacena en ZooKeeper la nueva ubicación de las nuevas regiones.

Algoritmo 5 Equilibrado de Datos Uniforme

Require: tabla: nombre de la tabla a equilibrar
precisionHistograma: precisión de histograma en número de filas
maxPorcentajeTuplasOtraRegión: porcentaje de tuplas que
están en una región que no le corresponde

```
1.  histograma ← generaHistograma(tabla, precisionHistograma)
2.  #RS ← get#RegionServers()
3.  #Regiones ← get#Regiones(histograma)
4.  #TuplasTabla ← get#Tuplas(histograma)
5.  #TuplasRegionEsperadas ← get#TuplasRegionEsperadas(histograma, #RS)
6.  porcentajeTuplasOtraRegión ← getPorcentajeTuplasOtraRegión(histograma,
    #TuplasRegionEsperadas
7.  if porcentajeTuplasOtraRegión > maxPorcentajeTuplasOtraRegión or
    #Regiones != #RS then
8.     puntosSplit ← getPuntosSplit(histograma, #TuplasRegionEsperadas
9.  else
10.    puntosSplit ← getPuntosSplit(tabla, #TuplasRegionEsperadas
11. end if
12. split(tabla, puntosDivision)
13. moveYmerge(tabla, puntosDivision)
14. majorCompact(tabla)
15. LocalizacionRegiones(tabla)
```

Una vez terminada la ejecución del Equilibrado de Datos Uniforme, todas las instancias HBase region server activas en el clúster tienen una región de la tabla y apro-

ximadamente todas ellas están sirviendo una cantidad de datos similar permitiendo asegurar que la latencia de acceso a los datos en todas las regiones es similar.

4.3. Evaluación del rendimiento

La evaluación del rendimiento del Equilibrado de Datos Uniforme se ha llevado acabo en un cluster de 11 máquinas AMD, cada nodo tiene 4 procesadores AMD Opteron 6376 @ 2.3GHz con un total de 64 cores virtuales, 128 GB de memoria RAM y todos ellos conectados mediante una red Ethernet a 1Gbit. Cada uno de los nodos tiene directamente conectado dos discos, un SSD (Intel SD3500 480GB) y un HDD con 2TB de capacidad y tienen instalado ubuntu 12.04.5 LTS. De las 11 máquinas, una de ellas es utilizada para ejecutar todos los servicios de metadatos, *nodo de control*, HDFS NameNode, HBase Master y ZooKeeper. El resto de máquinas van a tener un servicio HDFS DataNode encargado de almacenar los datos de HBase y 4 instancias del servicio HBase Region Server. De esta manera hay un total de 10 instancias HDFS DataNodes y 40 instancias HBase Region Servers en todo el cluster.

Tabla 4.1: Arquitectura de las máquinas utilizadas durante la evaluación

	AMD
Máquinas	11
Red	1Gb
Sockets	4 módulos x 1 sockets
Vcores	64
RAM (GB)	128
Discos	1 HDD + 1 SSD
Procesador	AMD Opteron 6376 @ 2.3GHz
Instancias HDFS-DN	10 (1 por máquina)
Instancias HBase-RS	40 (4 por máquina)

Para esta evaluación se ha utilizado la base de datos con un tamaño de 3000 Warehouses siguiendo con los resultados obtenidos en 3.9, en la tabla 4.2 se muestran para cada una de las tablas del benchmark TPC-C el tamaño en disco de los datos antes de ser almacenados en HBase y el número de tuplas que tiene cada una de ellas.

Tabla 4.2: Tamaño y número de tuplas TPC-C 3000 Warehouses

Nombre tabla	Tamaño	Número tuplas
Warehouse	254 KB	3000
District	2,7 MB	30000
Item	6,5 MB	100000
New_Order	303 MB	27M
Order	3,9 GB	90M
History	5,9 GB	90M
Customer	39 GB	90M
Stock	84 GB	300M
Order_Line	62 GB	765M
TOTAL	**169 GB**	*1663M*

Al realizar la carga de la base de datos se han creado 40 regiones con número de tuplas muy diferente para crear regiones no equilibradas. La tabla 4.3 muestra la información obtenida para cada una de las tablas de benchmark TPC-C. De cada una de las tablas se muestra el número de tuplas total, el número de tuplas que tiene la región con menos tuplas, el número de tuplas que tiene la región con más datos, el número de tuplas que se encuentran en una región que no le corresponde y el porcentaje que representa del número de tuplas que se encuentran en una región distinta. Por ejemplo de la tabla Warehouse con un total de 3000 tuplas, la región con menos tuplas tiene 6 tuplas, 234 tuplas tiene la región con más tuplas, en total hay 63 tuplas que no se encuentran en la región que les corresponde, lo que supone un 2,1 % del total de la tuplas de la tabla. La tabla con regiones más dispares en cuanto al número de tuplas es la tabla New_Order donde el número de tuplas colocadas en una región que no le corresponde es 619.998 de un total de 27 millones de tuplas lo que supone que un 22,59 % de las tuplas. Y por otro lado está la tabla History en la que solo un 0,19 % de las tuplas se encuentran en otra región, es decir 172.493 tuplas de 90 millones de tuplas que tiene la tabla en total. Se puede observar el número de tuplas por región en las tablas del anexo C.

El equilibrado de datos uniforme ha sido ejecutado sobre cada una de las tablas del benchmark TPC-C.Los parámetros utilizados en la ejecución del equilibrado para cada una de las tablas son la precisión del histograma (10000 tuplas) y un porcentaje

Tabla 4.3: Distribución de los datos de las tablas del benchmark TPC-C en 40 regiones.

Tabla	Número tuplas	Número tuplas región pequeña	Número tuplas región grande	Número tuplas otra región	Porcentaje tuplas otra región
warehouse	3000	6	234	63	2,1 %
district	30000	48	3267	795	2,65 %
item	100000	2	9388	22159	22,52 %
new_order	27M	17749	2837997	619998	22,66 %
orders	90M	33865	9721682	2306951	2,56 %
history	90M	2105753	2463901	172493	0,19 %
customer	90M	24592	7754305	2206718	2,45 %
stock	300M	18376	31182657	7463525	2,48 %
order_line	765M	11181	121495735	87312734	11,43 %

del número máximo de tuplas en regiones que no le corresponde (1 %). Con los valores obtenidos del histograma en la tabla 4.3, todas las tablas van a ser procesadas por el equilibrado de datos uniforme, a excepción de la tabla History, ya que el porcentaje de tuplas colocadas en una región que no les corresponden, 0,19 % es inferior al máximo indicado del 1 %.

En la tabla 4.4 se muestran los tiempos de ejecución de cada una de las fases del equilibrado de datos uniforme en segundos y para cada una de las tablas. Por ejemplo, para la tabla Customer con 90 millones de filas la generación del histograma ha tardado 95 segundos (algo más de minuto y medio), la obtención de los puntos de división de las regiones y la creación es la fase más costosa en todas las tablas, en este caso ha tardado 755 segundos (en torno a 12 minutos), la fase *move*Y*merge* que se encargan de agrupar las regiones iniciales con las nuevas para crear las regiones esperadas ha tardado 263 segundos. Una vez están las regiones creadas se realiza el *majorCompact*, para agrupar los HFiles de las regiones en HDFS, para la tabla Customer este proceso ha llevado unos 45 segundos y para finalizar la ubicación de las nuevas regiones en los distintos HBase Region Servers se guarda en ZooKeeper con un tiempo de 10 segundos. Finalmente, se muestra el tiempo total en realizar el equilibrado de la tabla que en este caso es de 1171 segundos, es decir unos 19 minutos.

El equilibrado de datos uniforme no se ha lanzado con la tabla History ya que el porcentaje de tuplas colocadas en una región que no les corresponde es de 0,19 %. Los

Tabla 4.4: Tiempo de ejecución (segundos) de cada una de las fases del Equilibrado de Datos Uniforme para las tablas del benchmark TPC-C.

Tabla	#Tuplas	Histograma	*Split*	*Move* y *Merge*	*Major Compact*	Ubicación regiones	Total
warehouse	3000	2	6	11	23	10	53
district	30000	2	8	9	2	10	54
item	100000	1	8	10	23	10	53
new_order	27M	21	46	17	23	10	117
orders	90M	88	470	72	45	10	685
history	90M	31	0	0	0	10	42
customer	90M	95	755	263	45	10	1171
stock	300M	309	2350	432	49	10	3151
order_line	765M	1136	6896	691	46	10	8781

tiempos mostrados en la tabla 4.4 para dicha tabla son: el tiempo de generación del histograma (31 segundos) y el tiempo del último paso del algoritmo 5 en el que se guarda la ubicación de las regiones en ZooKeeper (10 segundos).

La tabla Order_Line que tiene 87312734 tuplas colocadas en regiones que no le corresponden de un total de 756 millones, es decir el 11,43 %, el proceso de Equilibrado de Datos ha tardado casi 3 horas en finalizar siendo el paso más costoso la creación de nuevas regiones con 1136 segundos (1hora y 59 minutos).

Tras terminar el proceso de equilibrado en todas las tablas del benchmark TPC-C se ha vuelto a ejecutar el histograma para conocer la distribución de las tuplas en las distintas regiones. La tabla 4.5 muestra los resultados para cada una de las tablas. Las regiones de las tablas Warehouse, District e Item tras finalizar el Equilibrado de Datos Uniforme todas las regiones de cada una de las tabla tienen la 75, 750 y 2500 respectivamente. La tabla New_Order tenía un porcentaje de tuplas colocadas en una región que no le corresponde antes de ejecutar el equilibrado de datos del 22,66 %, después del 0,1 %, es decir de las 619.998 tuplas iniciales a 28.020 tuplas después del equilibrado. Y para el caso de la tabla más grande, la tabla Order_line con 765 millones de tuplas, el número de tuplas que pertenecen a una región que no les corresponde es de 29.743 tuplas lo que supone un 0,003 % del total de las tuplas. Por otro lado, la tabla History ya estaba equilibrada (0,19 %) por lo que no se ha ejecutado el equilibrado de

datos uniforme.

Tabla 4.5: Distribución de los datos de las tablas TPC-C después de ejecutar el Equilibrado de Datos Uniforme.

Tabla	Número tuplas	Número tuplas región pequeña	Número tuplas región grande	Número tuplas otra región	Porcentaje tuplas otra región
warehouse	3000	75	75	0	0 %
district	30000	750	750	0	0 %
item	100000	2500	2500	0	0 %
new_order	27M	500000	684408	28020	0,1 %
orders	90M	1980000	2260000	743379	0,8 %
history	90M	2105753	2463901	172493	0,19 %
customer	90M	1975165	2260000	44120	0,04 %
stock	300M	7202284	7510000	47762	0,01 %
order_line	765M	18940000	19134682	29743	0,003 %

4.4. Conclusiones

En este capítulo se ha presentado una herramienta que permite equilibrar los datos de las tablas almacenadas en HBase de manera uniforme entre todas las regiones. A continuación se muestra como mejora el tiempo de respuesta de acceso a los datos almacenados en HBase mediante la ejecución del benchmark TPC-C durante 20 minutos en el mismo despliegue en el que se ha realizado la evaluación con un *warm up* de 5 minutos y un *cold down* de 3 minutos. Este benchmark realiza un acceso uniforme a los datos almacenados y muestra su rendimiento teniendo en cuenta la cantidad de transacciones por minuto ejecutadas (throuhgput) y el tiempo medio en milisegundos que tarda en ejecutarse cada transacción (latencia). En una primera ejecución, los datos almacenados en HBase presentaban la distribución mostrada en la tabla 4.3, en el que en el peor de los casos una de las tablas tenía un 22,66 % de las tuplas colocadas en una región que no le correspondía. Con esta distribución de los datos tras la ejecución del benchmark se obtuvo un rendimiento de 3296 transacciones por minuto con una latencia media de 1550 milisegundos, columna “Regiones no equilibradas” de la tabla 4.6.

Tras haber ejecutado el Equilibrado de Datos Uniforme, el porcentaje de tuplas colocadas en una región que no le corresponde se encuentra por debajo de 1 % en todas las tablas del benchmark. Al ejecutar el benchmark con la distribución uniforme de los datos los resultados obtenidos han sido: 36761 transacciones por minuto con una latencia media de 16 milisegundos, 10 veces superior el número de transacciones por segundo y dos órdenes de magnitud menos en la latencia media en comparación con la ejecución del benchmark con los datos no equilibrados.

Tabla 4.6: Ejecución del benchmark TPC-C sobre una base de datos con regiones no equilibradas y con las regiones equilibradas.

	Regiones no equilibradas	Regiones equilibradas
Throughput (tpmCs)	3296	36761
Latencia media (ms)	1550	16

El Equilibrado de Datos Uniforme permite mejorar en 10 veces el rendimiento de una base de datos cuando se realizan consultas de acceso uniforme a los datos, haciendo que cada una de las regiones de las tablas almacenen la misma cantidad de datos. Este procesamiento solo se realiza cuando la distribución de los datos sobrepase un límite especificado o cuando el número de máquinas del clúster varíe asegurando tras su ejecución una distribución de los datos uniforme. Este trabajo fue presentado en la 20th International Conference on Extending Database Technology, con el título Load balancing for Key Value Data Stores [44].

Capítulo 5

Elasticidad Dinámica en UPM-CEP

La carga que recibe el CEP puede variar a lo largo del tiempo. La mayoría de las aplicaciones tienen un comportamiento en el que se producen muchos eventos a determinadas horas del día, algunos días y menos otros días. El despliegue de las consultas en el cep debería de adaptarse a la carga. Si hay más carga, serán necesarios más recursos. Si la carga disminuye, se liberarán recursos, es decir el sistema será elástico. La elasticidad es importante tanto en clouds públicas, donde se paga por uso, como en clouds privadas, ya que los recursos no necesarios puede ser apagados. La elasticidad deberá ser mínimamente invasiva, es decir, el sistema no parará el procesamiento, ni desplegará de nuevo las consultas, ya que durante ese tiempo el sistema no estará disponible y por tanto, no realizará procesamiento alguno. Este tiempo puede llevar varios minutos y las tuplas afectadas durante ese tiempo no serán procesadas. También puede darse el caso que el cep se encuentre distribuido en un entorno heterogéneo o que varias sub-consultas se estén ejecutando en el mismo nodo haciendo que los recursos del mismo no sea suficientes, para ello hay que migrar las sub-consultas de unos nodos a otros. Otro motivo de no disponibilidad es la caída de uno o más nodos donde está ejecutándose el sistema. Este escenario puede dar lugar a que dejen procesarse eventos de manera adecuada hasta que el sistema se vuelva a reestablecer y por ello dar lugar a la pérdida del procesamiento de grandes cantidades de eventos.

En este capítulo se van a presentar protocolos para dotar de tolerancia a fallos y elasticidad dinámica al UPM-CEP. El resto del capítulo está organizado de la si-

guiente manera: primero se va a mostrar una evaluación del sistema de tolerancia a fallos disponible en un sistema de procesamiento de datos en streaming, Flink [13] y se va a presentar un protocolo implementado para proporcionar tolerancia a fallos al componente Cep-instanceManager de UPM-CEP (sección 5.1). A continuación se va a mostrar el protocolo e implementación del proceso de migración de sub-consultas entre distintos Cep-instanceManagers (sección 5.2), seguido se va a presentar el protocolo e implementación que permite añadir instancias de una sub-consulta y finalmente el protocolo e implementación de quitar instancias de una sub-consulta sin detener el flujo de los datos en ambos casos, (secciones 5.3 y 5.4). Para finalizar se va a mostrar la evaluación del rendimiento de cada una de las contribuciones y se va a demostrar la capacidad de escalar del UPM-CEP, sección 5.5.

5.1. Tolerancia a Fallos

Los sistemas que permiten procesar flujos continuos de eventos poseen diversos sistemas de tolerancia a fallos. Estos les permiten reconfigurar el sistema en caso de que se detecte un fallo en alguno de los nodos que procesan eventos. Un ejemplo es Flink el cual tiene un sistema de tolerancia a fallos basado en *checkpointing*. En esta sección se va a evaluar el rendimiento de la tolerancia a fallos de Flink y proponer protocolos alternativos con menor coste en términos de latencia cuando curre un fallo. Estos protocolos se han incorporado en UPM-CEP.

5.1.1. Checkpointing

Flink posee un sistema de tolerancia a fallos basado en fuentes de datos fiables y puntos de control de estado [45], *checkpointing*. Un data source fiable lee los eventos desde un sistema de mensajería como Kafka [46] o RabbitMQ [47]. De esta manera en caso de fallo es posible mandar de nuevo los eventos desde un punto específico anterior en el tiempo. Cuando el sistema de *checkpointing* está activado, cada X eventos procesados se emite una tupla de control llamada *barrier*. Cada *barrier* tiene un identificador único y éste es incluido en el stream como un evento más del data source. Dependiendo

del tipo de operador, con estado o sin estado, el comportamiento al recibir los *barriers* es distinto. Si el operador sin estado únicamente tiene un stream de entrada, todos los eventos entre un *barrier-n* y el siguente *barrier-n+1* son considerados como parte del *barrier-n*. En caso de que tenga más de un stream de entrada, el operador retiene los eventos recibidos hasta que recibe el *barrier-n* de todos los streams. En ese momento comienza a procesar todos los eventos en el orden recibido. De esta manera, en caso de fallo Flink solicitará recibir de nuevo los eventos posteriores a *barrier-n*. En cambio, los operadores con estado, escriben los eventos almacenados en las ventanas en HDFS. Los eventos almacenados en las ventanas entre dos *barriers*, *barrier-n* y *barrier-n+1*, se almacenan en HDFS según son añadidos en la respectiva ventana. Una vez el operador recibe el *barrier-n+1*, los eventos almacenados en HDFS pueden ser eliminados, ya que todos los eventos pertenecientes al *barrier-n* ya han sido procesados. De esta manera si se produce un fallo, se puede recuperar el estado de las ventanas pertenecientes al *barrier-n* que han sido almacenados en las ventanas hasta el momento del fallo.

En caso de que ocurra un fallo, se redespliega la consulta en las instancias que haya disponibles en ese momento, se recupera el estado en los operadores con estado en el nuevo despliegue y se solicita a los sistemas de mensajería que reenvíen los eventos enviados antes de producirse el fallo. Una vez se han terminado de procesar los eventos pertenecientes al *barrier-n*, se procesan los eventos almacenados durante la recuperación del sistema ante el fallo y seguido se reestablece el flujo de datos normal. Durante esta reconfiguración, el sistema deja de procesar eventos bloqueando el correcto procesamiento de los mismos.

Se ha evaluado el rendimiento de Flink cuando falla uno de los nodos, además del coste que introduce el sistema de *checkpointing* en el procesamiento de los eventos. Para ello se ha utilizado una variante de la consulta FixedTimewindow del benchmark HiBench, en la cual en vez de utilizar ventanas de tiempo se utilizan ventanas de tamaño (50 tuplas), y se ha desplegado Flink junto a Kafka y HDFS, como sistemas de mensajería y de almacenamiento persistente de datos. La evaluación se ha realizado en un clúster de seis máquinas XEON cada una de ellas equipada con dos procesadores

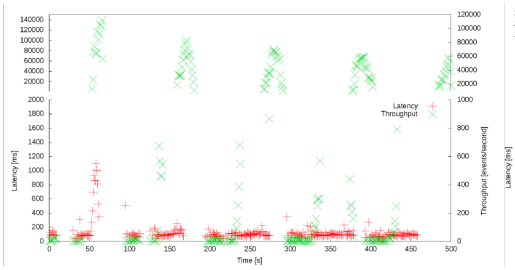


Figura 5.1: Resultados de la evaluación sin *checkpointing*

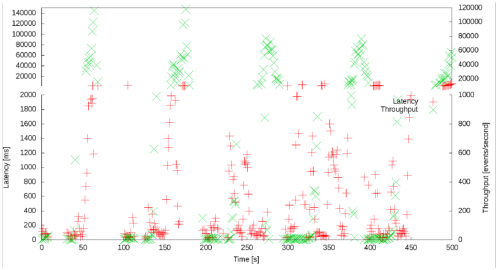


Figura 5.2: Resultados de la evaluación con *checkpointing*

Intel XEON E5-2620 v3 con 12 cores virtuales (24 cores virtuales en total) y 128 GB de memoria RAM. Todas ellas están conectadas mediante una red Ethernet a 1 Gbit. En una de las máquinas se ejecutan los clientes que inyectan la carga al sistema. Una segunda máquina tiene en ejecución el proceso máster de Flink, junto con ZooKeeper y HDFS. Las máquinas tres y cuatro tienen en ejecución cada una de ellas 6 instancias del sistema de mensajería Kafka. Y las máquinas cinco y seis tienen 12 instancias de Flink cada una, donde se procesan los eventos obtenidos de las instancias Kafka.

Para ello se ha comparado el rendimiento sin el mecanismo *checkpointing* y con el mecanismo activado, en ambos casos se ha medido el tiempo de procesamiento de los eventos en milisegundos (latencia) y el número de eventos procesados al segundo (throughput). En todo momento la carga enviada a Kafka a través de los clientes es de 500.000 eventos al segundo. Las figuras 5.1 y 5.2, muestran los resultados obtenidos en ambas evaluaciones, donde se puede observar que la latencia en la evaluación sin *checkpointing* es inferior a 200 milisegundos, mientras que con *checkpointing* asciende hasta los 2 segundos cuando se almacena el estado en HDFS. El throughput observado en ambas ejecuciones es de 100.000 eventos al segundo y es alcanzado hasta en 5 ocasiones, esto es debido a quel el número de claves de las ventanas permanece estable durante la evaluación y la carga enviada hace que las ventanas se llenen con mayor facilidad produciendo que estas se deslicen y generen nuevos eventos de salida. Por lo tanto, la activación del *checkpointing* muestra un coste elevado produciendo latencias 10 veces superiores a las obtenidas en el mismo escenario sin el *checkpointing* activado.

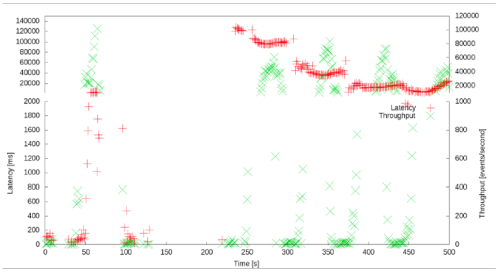


Figura 5.3: Resultados de la evaluación con *checkpointing* y fallo

También se ha evaluado el comportamiento del sistema cuando ocurre un fallo. En este caso, se ha parado una de las instancias de Flink a los 90 segundos de la ejecución. La figura 5.3 muestra la latencia y el throuhgput durante la evaluación. El sistema tarda 90 segundos en detectar que ha ocurrido un fallo, reconfigurar el sistema, cargar el estado almacenado en HDFS y comenzar con el procesamiento normal de los eventos. Durante este periodo el sistema deja de procesar eventos y una vez reconfigurado tarda hasta 260 segundos en procesar los eventos que no habían terminado de ser procesados cuando ha ocurrido el fallo y en procesar los eventos que estaban esperando en Kafka a ser procesados durante la reconfiguración, mostrando latencias de 20 segundos.

Como conclusión, se ha observado que el mecanismo de tolerancia a fallos *checkpointing* tiene un elevado coste en términos de latencia (10 veces más). Además durante cerca de minuto y medio dejan de procesarse eventos, lo cual produce que se acumulen hasta 45 millones de eventos en el servicio de mensajería Kafka a ser procesados una vez el sistema termine de reconfigurarse. Tras la reconfiguración, el sistema no termina de estabilizarse y procesar eventos con una latencia cercana a la obtenida antes de que ocurriese el fallo durante el tiempo transcurrido en la evaluación.

5.1.2. Replicación Activa

Con el objetivo de evitar el bloquedo del sistema durante el proceso de reconfiguración ante fallos se ha implementado un protocolo de tolerancia a fallos basado en replicación activa. Este protocolo consiste en desplegar varias réplicas del mismo proceso, las cuales van a recibir los mismos datos en el mismo orden, para ello se emplea

un radiado atómico [48] y [49]. El principal objetivo de la implementación del protocolo de replicación activa en UPM-CEP es el hacer el componente instanceManager (IM) tolerante a fallos. Para ello se ha ampliado la funcionalidad de los operadores DataSource y DataSink de tal manera que el primero de ellos envía con radiado atómico las tuplas que recibe del cliente y el segundo analiza las tuplas que recibe de tal manera que solo envía al cliente la tupla procedente de la réplica más rápida. En caso de fallo de una de las réplicas (IM), el DataSource deja de enviar tuplas a la réplica y el DataSink envía al cliente las tuplas de la réplica activa. De esta manera se evita que el sistema se detenga durante la reconfiguración ante el fallo.

La figura 5.4 muestra el flujo de las tuplas a través del UPM-CEP cuando se ha desplegado la consulta con replicación activa. En este escenario, la consulta está formada por una sub-consulta y ésta está replicada en dos nodos. El nodo 1 tiene los servicios orchestrator, metriserver y ZooKeeper y los clientes que envían y reciben las tuplas (Cliente1 y Cliente2). El nodo 2 tiene tres instanceManagers (IM1, IM2 e IM3) donde el primero de ellos tiene el operador DataSource, el segundo tiene la Sub-Consulta 1 y el tercero tiene el operador DataSink. El nodo 3, tiene instanceManager (IM4) con otra instancia o réplica de la Sub-Consulta 1. Las tuplas llegan al operador DataSource, les asigna un identificar numérico y duplica el flujo enviándolo a ambas réplicas de la sub-consulta 1. Cada una de ellas, tras procesar las tuplas, las envía al operador DataSink donde almacena los identificadores de las tuplas recibidas. Si es la primera vez que llega dicha tupla, la envía al cliente que esté recibiendo las tuplas. Si ya ha recibido una tupla con el mismo identificador, descarta la tupla y elimina el identificador de la lista.

En caso de que falle alguna de las dos réplicas, el operador DataSink lo detecta ya que pierde la conexión con la réplica fallida. En ese momento comienza a comparar las tuplas que llegan de la réplica que sigue en funcionamiento con la lista de identificadores, eliminando los que lleguen repetidos hasta que el identificador de las tuplas que llegan sea mayor que los que tiene almacenados. A partir de ahí, el operador DataSink comienza a funcionar de la manera habitual enviando las tuplas al cliente según las va recibiendo sin realizar ningún procesamiento con ellas.

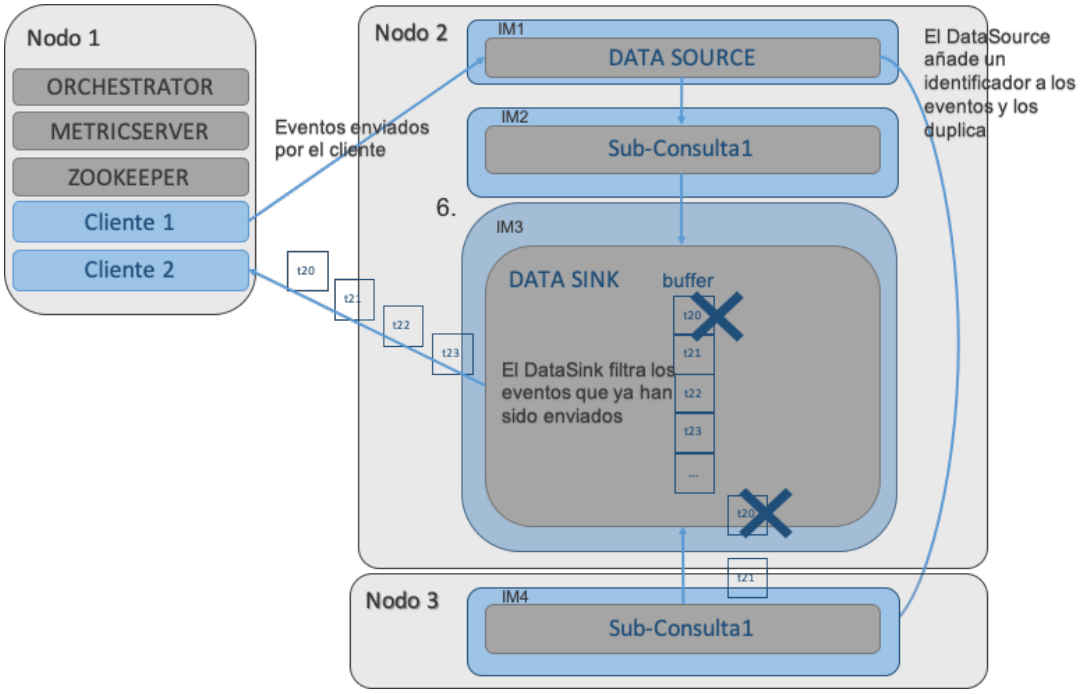


Figura 5.4: Funcionamiento de la Replicación Activa en una consulta compuesta por una sub-consulta

5.2. Migración

El proceso de migración consiste en mover una instancia de una sub-consulta de un Cep-instanceManager (IM) a otro. Este proceso se lanza de manera automática cuando el consumo de CPU es muy alto o la cantidad de memoria libre es muy baja (75 % CPU y 500 MB RAM libre) y en el mismo IM hay en ejecución: 1) Un operador DataSource o DataSink más una o varias sub-consultas, 2) más de una sub-consulta y 3) solo hay una sub-consulta y hay otros IMs con más recursos (RAM) que el IM en el que está corriendo la sub-consulta.

El proceso de migración actúa de diferente manera si la sub-consulta a migrar tiene estado o no. Si no tiene estado, se despliega la sub-consulta en el nuevo IM y se modifica el balanceador del stream superior para que dirija las tuplas a la nueva instancia antes de eliminar la instancia inicial del IM. En cambio, si la sub-consulta tiene estado, las tuplas que se están almacenando en la instancia inicial tienen que ser enviadas

nueva instancia. El proceso, transferencia de estado, es muy costoso y llega a ocupar aproximadamente el 85 % del tiempo que tarda en ejecutarse la migración.

Una vez que se ha desplegado la sub-consulta en el nuevo IM, comienza la transferencia de estado. En ese momento el balanceador del stream superior ha dejado de enviar tuplas a la instancia a migrar y las está almacenando en un buffer interno para enviarlas al terminar la transferencia de estado. Tanto las ventanas de tiempo como las de tamaño han dejado de recibir tuplas y por lo tanto no se desplazan. Esto es debido a que las ventanas de tiempo se desplazan teniendo en cuenta un campo timestamp, si la diferencia de ese campo entre la primera tupla almacenada en la ventana y la recibida es igual o superior al tamaño de la ventana esta se desplaza. Las ventanas de tamaño se desplazan cuando ha llegado a almacenar tantas tuplas como el tamaño de las ventanas. Cuando se va a realizar la transferencia de estado, se comienza a recorrer todas las ventanas almacenadas en el operador con estado de las sub-consulta y cada una de las tuplas se envía a la nueva instancia. La nueva instancia comienza a recibir tuplas y va creando ventanas y almacenando las tuplas según sea necesario. Una vez que la instancia inicial ha terminado de enviar las tuplas, avisa al Cep-orchestrator y este se queda a la espera de que la nueva instancia le indique que ha terminado de procesar las tuplas que le ha enviado la instancia inicial. Momento en el que se da por finalizado el proceso de transferencia de estado.

La figura 5.5 muesta el proceso de migración de una sub-consulta (Sub-Consulta1) que se encuentra en ejecución junto con el operador DataSource en el mismo IM (IM1). Inicialmente el despliegue constan de dos IMs (IM1 e IM2) ambos desplegados en el nodo 2 del clúster UPM-CEP. Este nodo tiene capacidad suficiente para desplegar más IMs. En el nodo 1 se encuentran el Orchestrator, MetricServer, Zookeeper y los clientes que envían y reciben tuplas del UPM-CEP, cliente 1 y cliente 2, respectivamente. Tras un tiempo la carga recibida por la consulta se incrementa y la distribución actual no es capaz de procesarla. El orchestrator analiza las métricas de consumo de CPU y memoria que le envía el MetricServer de los IMs desplegados y observa que el consumo de CPU del IM1 es superior al 75 % y que en ese IM se están ejecutando un operador DataSource y la Sub-Consulta 1. El orchestrator lanza un nuevo IM (IM3) seleccionado

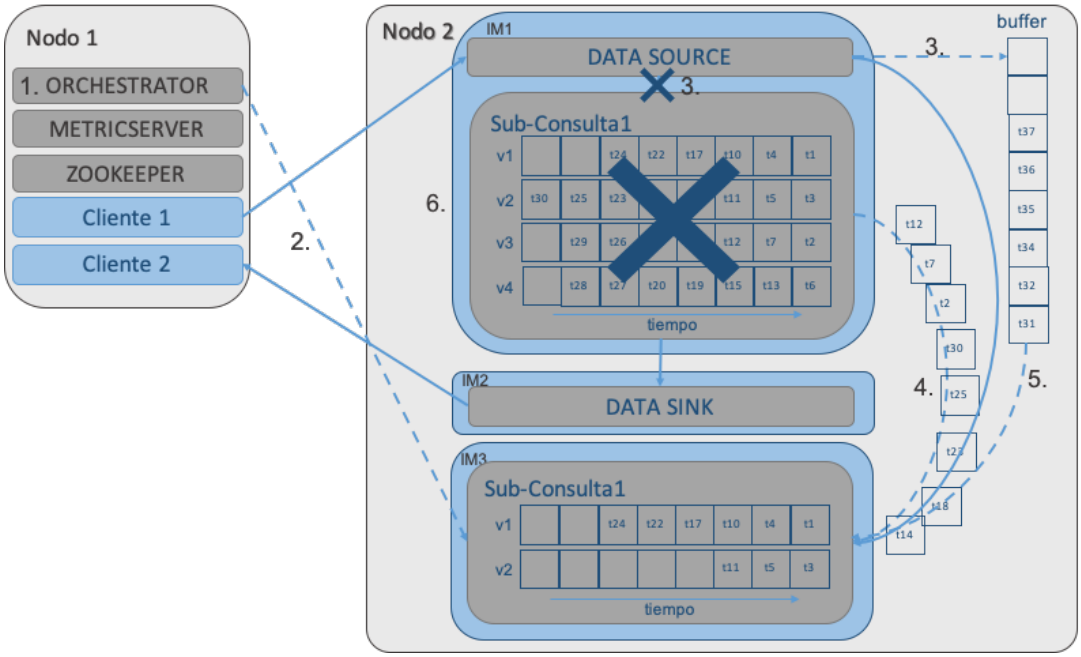


Figura 5.5: Migración sub-consulta con estado

del pool de posibles IMs a lanzar, comienza con el proceso de migración (paso 1). En el algoritmo 6, se muestra en detalle los pasos del proceso de migración. El orchestrator envía al IM3 toda la información para registrar y desplegar la Sub-Consulta 1 (paso 2, figura 5.5 y línea 1, algoritmo 6). Durante este proceso la consulta sigue procesando las tuplas recibidas a través de la Sub-Consulta 1 en el IM1, hasta que se pausa el stream del balanceador del DataSource y este comienza a almacenar las tuplas recibidas (t31, t32, ... en el buffer) (paso 3, figura 5.5 y líneas 2 y 3, algoritmo 6). El buffer en el que se almacenan las tuplas es limitado para no dejar al IM sin memoria. En caso de que se llene, las nuevas tuplas son descartas y no serán procesadas nunca.

En el ejemplo de la figura 5.5, la Sub-Consulta1 tiene estado por lo que se lanza el proceso de transferencia de estado desde la Sub-Consulta1 en el IM1 a la Sub-Consulta1 en el IM3 (paso 4, figura 5.5 y líneas 4 - 7, algoritmo 6). En la figura 5.5 se muestra la Sub-Consulta en el IM3 que ya ha recibido y procesado las tuplas t1 a t 11 creando las ventanas v1 y v2, correspondientes a las tuplas recibidas. Cuando termina

Algoritmo 6 Migración de una instancia de una sub-consulta

Require: nuevoIM: identificador del Cep-instanceManager (IM) que va a recibir la instancia
Require: consulta(C): nombre de la consulta que contiene la instancia de la subquery a migrar
Require: sub-consulta (SC): nombre de la subconsulta que contiene la instancia a migrar
Require: sub-consulta-inst (SCI): instancia de la subconsulta a migrar
1: registroDepliegueSCI-IM(sub-consulta-inst, nuevoIM)
2: *streams*Entrada ← obtenerInfo*Streams*EntradaSC(infoDespliegue, sub-consulta)
3: pararFlujoTuplas*Streams*Entrada(*streams*Entrada)
4: **if** sub-consulta con estado **then**
5: transferirEstado(sub-consulta-inst)
6: esperarTransferirEstado()
7: **end if**
8: reiniciarFlujoTuplas*Streams*Entrada(*streams*Entrada)
9: eliminarSCI-IMInicial(infoDespliegue)

este proceso, se modifica el balanceador del DataSource para que empiece a enviar las tuplas almacenadas durante el proceso de transferencia de estado a la Sub-Consulta1 en el IM3 y se elimina la Sub-Consulta1 del IM1 (paso 5, figura 5.5 y líneas 8 y 9, algoritmo 6). Momento en el que se da por finalizado el proceso de migración. En caso de que haya más instancias de la sub-consulta, estas siguen procesando las tuplas mientras se migra la sub-consulta indicada. Una vez termina el proceso de migración la consulta comienza a procesar toda la carga que está recibiendo.

5.3. Aumentar el número de instancias

El proceso scale up y scale out permite aumentar el número de instancias de las sub-consultas en un mismo nodo o en otro nodo del clúster, respectivamente. Este proceso se lanza de manera automática cuando el consumo de recursos (CPU y RAM) del IM en el que se está ejecutando la sub-consulta es alto (más 75 % CPU y menos 500 MB libres).

El proceso funciona de la misma manera cuando es scale up y cuando es scale

out. La principial diferencia se da si la sub-consulta a escalar tiene estado o no. Si la sub-consulta no tiene estado, se despliega la nueva instancia en un IM que puede estar en el mismo nodo que la instancia que se encuentra saturada o en otro nodo distinto. Una vez se ha desplegado la instancia se modifica el balanceador del stream o streams superiores de tal manera que se distribuye la carga entre ambas instancias.

Por otro lado si la sub-consulta es con estado, se realiza el proceso de transferencia de estado pero solo de la mitad de las ventanas, si todas ellas están recibiendo la misma carga o de algunas ventanas en concreto si hay algunas que reciben más carga que otras. El objetivo es balancear la carga para que la instancia saturada y la nueva reciban la misma cantidad de carga.

La figura 5.6 muestra un ejemplo de scale up en el cual la sub-consulta 1 instancia 1 (Sub-Consulta1-1) que se está ejecutando en el IM2 se escala dando lugar al despliegue de una instancia nueva, Sub-Consulta1-2 en el IM4. Inicialmente el despliegue constan de tres IMs (IM1, IM2 e IM3) todos desplegados en el nodo 2 del clúster UPM-CEP. Este nodo tiene capacidad suficiente para desplegar más IMs. En el nodo 1 se encuentran el Orchestrator, MetricServer, Zookeeper y los clientes que envían y reciben tuplas del UPM-CEP, cliente 1 y cliente 2, respectivamente. La consulta desplegada en este clúster de dos nodos consta de una sub-consulta con estado, el tipo de operador con estado no interfiere en el proceso de escalado de la sub-consulta. En los instanceManagers IM1 e IM3 se encuentran desplegados los operadores DataSource y DataSink y en el IM2 se encuentra desplegada la Sub-Consulta1-1. Tras un tiempo la carga a procesar se incrementa y la Sub-Consulta1-1 no es capaz de procesarla. El orchestrator al comprobar mediante las métricas del consumo de recursos de los IMs observa que el IM2 se encuentra saturado ya que la cantidad de memoria libre es inferior a 500MB. En ese momento el orchestrator despliega un nuevo IM, IM4 y comienza con el proceso de escalado de la Sub-Consulta1-1 (paso 1). En un primer momento envía la información necesaria al IM4 para que despliegue una nueva instancia de la Sub-Consulta1, Sub-Consulta1-2 (paso2). Una vez se ha desplegado la instancia, se configura el stream de salida de dicha instancia para que pueda enviar las tuplas de salida al operador DataSink que se encuentra en el IM3 (paso 3). Se configura

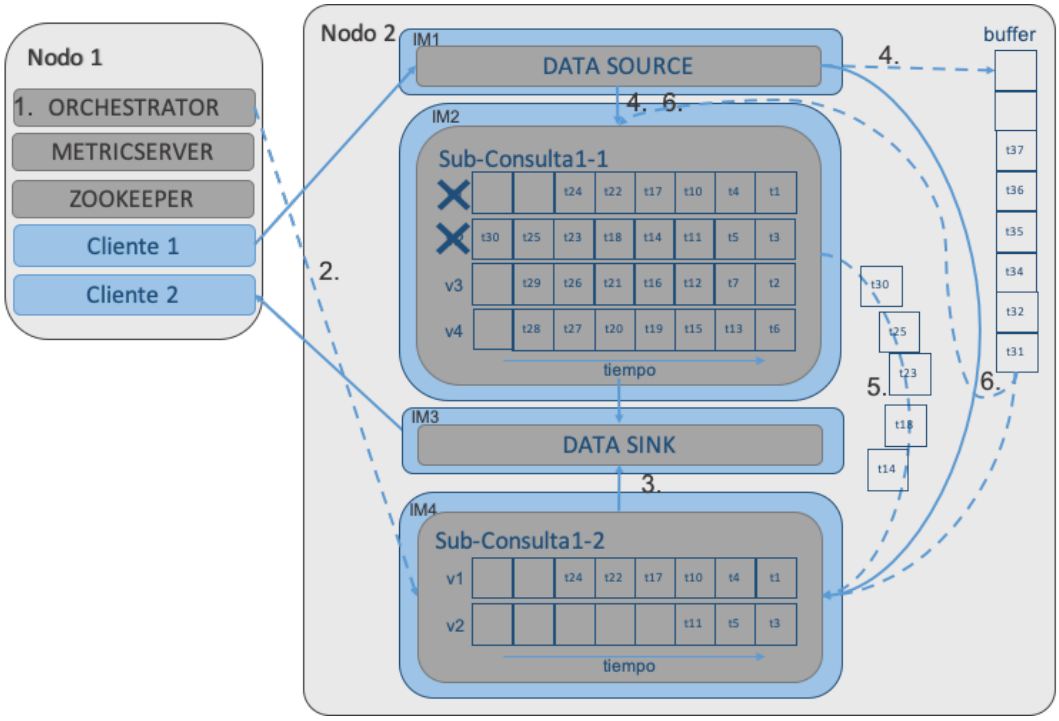


Figura 5.6: Escalado sub-consulta

el balanceador *GroupKey* del stream de salida del operador DataSource de tal manera que se pausa el envío de tuplas a la instancia que se está escalando y las almacena en el buffer (paso 4). Si hubiese más instancias de la Sub-Consulta1 en ejecución en el clúster estas seguirían recibiendo tuplas a procesar.

A continuación comienza el proceso de transferencia de parte de las ventanas, en este caso todas ellas reciben la mima carga (ventanas v1 a v4 en Sub-Consulta1-1) por lo que la mitad de las ventanas (v1 y v2) son transferidas a la instancia Sub-Consulta1-2 en el IM4 (paso 5). Cuando ha terminado el proceso de transferencia, la Sub-Consulta1-1 elimina las ventanas v1 y v2 y la Sub-Consulta1-2 ha terminado de procesar las tuplas de las ventanas. En este momento se configura el balanceador del stream de salida del operador DataSource, para que redirija las tuplas de las ventanas v1 y v2 a la nueva instancia y se reestablece el flujo de tuplas comenzando con las almacenadas en el buffer durante el proceso scale up. Una vez terminado este proceso, la consulta es capaz de procesar la carga de tal manera que la carga que estaba

soportando la Sub-Consulta1-1 ha sido distribuida entre ella y la Sub-Consulta1-2.

El algoritmo 7 muestra paso a paso el funcionamiento del proceso de escalado válido tanto para scale up como scale-out.

Algoritmo 7 Escalado de una instancia de una sub-consulta

Require: nuevoIM: identificador del Cep-instanceManager (IM) que va a recibir la nueva instancia
Require: consulta(C): nombre de la consulta que contiene la instancia de la subquery a escalar
Require: sub-consulta (SC): nombre de la subconsulta que contiene la instancia a escalar
Require: sub-consulta-inst (SCI): instancia de la subconsulta a escalar
Require: buckets: instancia de la subconsulta a migrar
1: infoDespliegue ← obtenerInfoDespliegue(consulta)
2: infoSub-Consulta ← obtenerInfoSC(infoDespliegue, sub-consulta)
3: nuevaSCI ← obtenerNuevoIdentificador
4: registrarSCI-IM(nuevaSCI, nuevoIM, infoSub-Consulta)
5: desplegarSCI-IM(nuevaSCI)
6: modificarSCI-Despliegue(infoDespliegue, nuevaSCI, nuevoIM)
7: *streams*Salida ← obtenerInfo*Streams*SalidaSC(infoDespliegue, sub-consulta)
8: modificarBalanceador*streams*Salida(*streams*Salida)
9: *streams*Entrada ← obtenerInfo*Streams*EntradaSC(infoDespliegue, sub-consulta)
10: pararFlujoTuplasYConfigurar*Streams*Entrada(*streams*Entrada,buckets)
11: **if** sub-consulta con estado **then**
12: transferirEstado(sub-consulta-inst, buckets)
13: esperarTransferirEstado()
14: **end if**
15: reiniciarFlujoTuplas*Streams*Entrada(*streams*Entrada)

El orchestrator comienza con el proceso de escalado de la sub-consulta saturada y para ello obtiene la información necesaria de la instancia de la sub-consulta a desplegar, tanto los operadores y streams que forman la sub-consulta como el IM que tiene desplegada la instancia a escalar (líneas 1 y 2, algoritmo 7). A continuación se registra y se despliega la sub-consulta en el IM indicado asignándole un nuevo identificador (líneas 3 - 5). Una vez la instancia de la sub-consulta se ha desplegado se registra la

ubicación de la nueva instancia en el ZooKeeper (línea 6). Seguido, se configura el balanceador del operador de salida para que se prepare para recibir tuplas desde el *stream* de salida de la instancia anterior, este proceso se realiza por cada uno de los *streams* de salida de la sub-consulta que envíen los datos a un operador de salida (líneas 7 y 8). Se detiene el envío de eventos desde el balanceador del stream o streams de entrada a la sub-consulta y se configura con la nueva distribución de las tuplas (líneas 9 y 10). En caso de que la sub-consulta sea sin estado el balanceador va a ser *round-robin* por lo que con solo añadir la nueva instancia al balanceador ésta ya está preparado. En cambio, si la sub-consulta es con estado el balanceador es *GroupKey*. En este caso si no se ha especificado que buckets van a ser servidos por la nueva instancia, se distribuyen los buckets de la instancia que se está escalando de manera uniforme entre la instancia escalada y la nueva instancia y en caso de que se indiquen los buckets se modifica el array de buckets indicado el identificador de la nueva instancia para los buckets indicados. Si la sub-consulta es con estado, se envían las tuplas de los buckets que van a ser servicios por la nueva instancia, líneas 11 a 14. Y finalmente se restablece el flujo de tuplas de los balanceadores de los *streams* de entrada, donde se comienza enviando las tuplas que han sido almacenados durante el proceso de traspaso del estado y continuando con el flujo normal, línea 15.

5.4. Reduciendo el número de instancias

El proceso scale-down permite reducir el número de instancias de una sub-consulta y a su vez liberar recursos cuando se observa que la carga que procesan todas las instancias es baja y que la misma carga puede ser procesada si se elimina una de las instancias. El principal requisito para poder ejecutar el *scale-down* es que al menos haya dos instancias de la sub-consulta y al igual que ocurre con el resto de protocolos que hacen que el UPM-CEP sea elástico el proceso varía dependiendo de si la sub-consulta es con estado o sin estado.

Si la sub-consulta es sin estado, cuando el orchestrator detecta que la sub-consulta puede reducir el número de instancias, este escoge la sub-consulta que menos carga

está procesando o en caso de que todas estén procesando la misma cantidad la escoge de manera aleatoria entre todas las disponibles. A partir de ese momento, se pausa el envío de tuplas a las sub-consultas a través del balanceador del stream o streams superiores. Seguido se modifica la configuración de los balanceadores de tal manera que ya no se van a enviar tuplas a la instancia que se está eliminando y se vuelve a recuperar el procesamiento de tuplas, primero comenzando con las tuplas almacenadas durante el proceso de scale-down y luego siguiendo con el flujo normal. Para finalizar se elimina la sub-consulta dejando el IM libre o con la ejecución de otra sub-consulta en caso de que hubiese más de una sub-consulta ejecutándose.

La figura 5.7 muestra un ejemplo en el que la sub-consulta1 con estado tiene dos instancias donde se va a eliminar la instancia 2 de la sub-consulta1 para que sea la instancia 1 la encargada de procesar todos las tuplas. En el ejemplo hay un clúster con dos nodos (nodo 1 y nodo 2) donde el nodo 1 tiene el Orchestrator, MetricServer, Zookeeper y los clientes que envían y reciben tuplas del UPM-CEP, cliente 1 y cliente 2, respectivamente. El modo 2 tiene cuatro IMs (IM1, IM2, IM3 e IM4), los IMs IM1 e IM4 tiene los operadores DataSource y DataSink, respectivamente. Los IMs IM2 e IM3 tiene las instancias Sub-Consulta1-1 y Sub-Consulta1-2. Una vez el orchestrator selecciona la instancia de la Sub-Consulta1 a eliminar comienza el proceso scale-down (paso 1). En un primer momento, se detiene el flujo de tuplas del balanceador del *stream* de salida del operador DataSource y se configura eliminando la configuración de la instancia a eliminar (paso 2). A continuación, hace la transferencia de estado enviando las ventanas de la Sub-Consulta1-2 (v1 y v2) a la Sub-Consulta1-1 (paso 3). En caso de que hubiese más instancias de la Sub-Consulta1 las ventanas se distribuirían de manera uniforme entre las instancias disponibles para balancear la carga una vez termine el proceso de scale-down. Una vez terminada la transferencia de estado, se restablece el flujo de tuplas entre las instancias restantes comenzando por el envío de las tuplas almacenadas (paso 4). Y finalmente se elimina la Sub-Consulta1-2 del IM3 y se guarda la nueva configuración en ZooKeeper.

El algoritmo 8 presenta los pasos a seguir para eliminar una instancia de una sub-consulta. El orchestrator obtiene la información de despliegue de la consulta y con

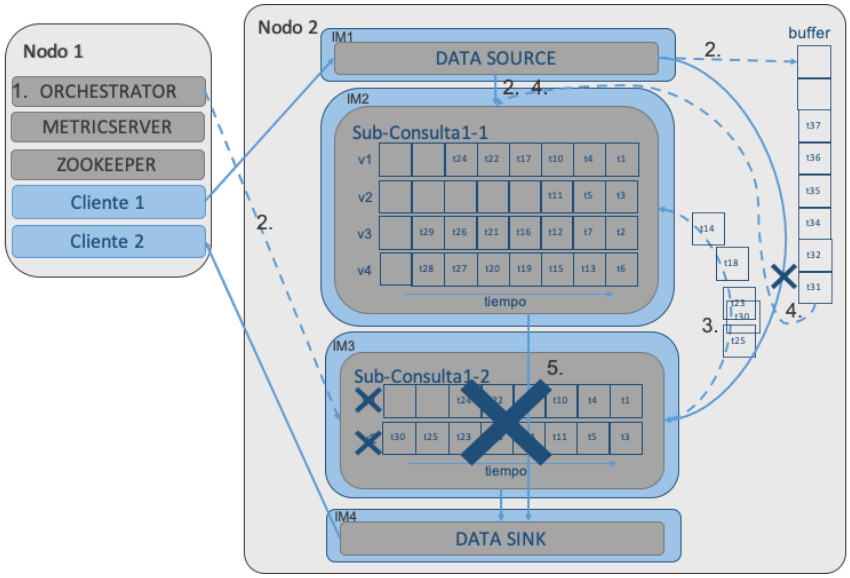


Figura 5.7: scale-down sub-consulta

ello, el IM en el que está en ejecución la instancia a eliminar (línea 1). A continuación, se obtienen los balanceadores de los *streams* de entrada para configurar y detener el envío de tuplas a las instancias de la sub-consulta (líneas 2 y 3). Seguido, en caso de que la sub-consulta sea con estado, se envían las tuplas almacenados en la instancia a eliminar entre las instancias que sigan activas de la sub-consulta de manera uniforme (líneas 4 - 7). Finalmente se restablece el flujo de los balanceadores de los *streams* de entrada, se elimina la instancia del IM y se almacena la información del despliegue actual de la sub-consulta en ZooKeeper (líneas 8 y 9).

5.5. Evaluación del rendimiento

La evaluación de las aportaciones de este capítulo se ha realizado sobre tres despliegues diferentes: 1) Clúster AMD con tres nodos conectados mediante una red Ethernet de 1 Gb, 2) Clúster Intel XEON con cinco nodos conectados mediante una red Ethernet de 1 Gb y 3) dos nodos multicore, Bullion Sequana S800 conectados mediante una red Ethernet de 10 Gb.

Algoritmo 8 Scale-Down de una instancia de una sub-consulta

Require: consulta(C): nombre de la consulta que contiene la instancia de la subquery a eliminar
Require: sub-consulta (SC): nombre de la sub-consulta que contiene la instancia a eliminar
Require: sub-consulta-inst (SCI): instancia de la sub-consulta a eliminar
1: infoDespliegue ← obtenerInfoDespliegue(consulta)
2: *streams*Salida ← obtenerInfo*Streams*SalidaSC(infoDespliegue, sub-consulta)
3: pararFlujoTuplas*Streams*Entrada(*streams*Entrada)
4: **if** sub-consulta con estado **then**
5: transferirEstado(sub-consulta-inst)
6: esperarTransferirEstado()
7: **end if**
8: reiniciarFlujoTuplas*Streams*Entrada(*streams*Entrada)
9: eliminarSCI-IM(infoDespliegue)
10: modificarSCI-Despliegue(infoDespliegue, SCI)

Para evaluar cada una de las funcionalidades de ha utilizado el Benchmark HiBench [51]. HiBench es una suite que permite evaluar el rendimiento distintos sistemas de streaming como Spark Streaming, Flink y Storm. En concreto se ha seleccionado la consulta *Fixed Time Window*, la cual permite evaluar el rendimiento de las ventanas, en concreto mediante el operador aggregate, esta ha sido implementada y desplegada en el UPM-CEP. La consulta *Fixed Time Window* procesa conexiones recibidas a un sitio web desde diferentes IPs y muestra cada X segundos cuantas conexiones se han realizado desde cada una de las IPs. La consulta consta de 5 operadores, tal y como se muestra en la figura 5.8: 1) Operador SocketDataSource, encargado de recibir las tuplas producidas cada vez que se realiza una conexión a través de un socket. Los tuplas tienen el siguiente formato: <*userTS*:long, IP:String, *sessionId*:String, *browser*:String, *payload*:byte>, el campo userTS es la marca de tiempo que se le asigna al evento y tiene el momento en el que se ha realizado la conexión con la página web. Las IPs se generan de manera uniforme desde 255.255.0.0 a 255.255.255.255 generando un total de 65536 IPs diferentes. El campo *sessionId* es un String formado por el String "sess-" seguido de la dirección IP. El campo *browser* es un String que puede tener el valor *chrome*, *ie*, *firefox*, *opera* o *safari* y el campo *payload* es un array de 4 bytes aleato-

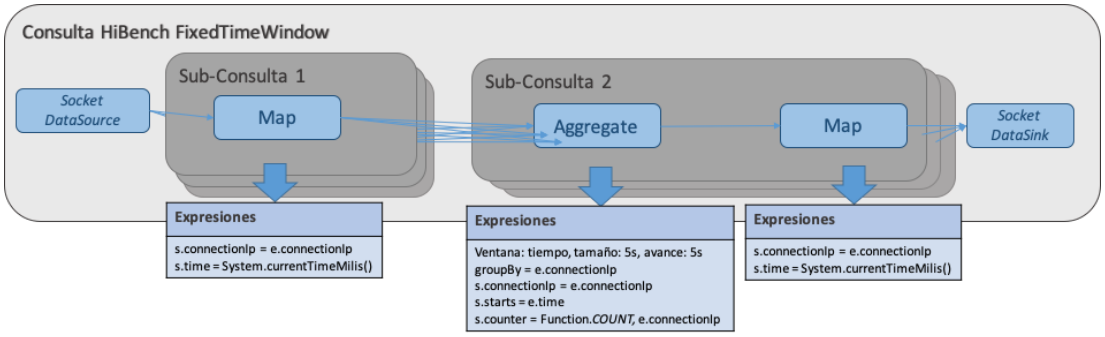


Figura 5.8: Consulta Fixed Time Window del Benchmark HiBench

rios. 2) Operador Map, filtra los campos de las tuplas que recibe y únicamente deja los campo IP y browser, es decir las tuplas de salida del operador Map tienen el siguiente esquema: <*userTS*:long, IP:String, *payload*:byte>. 3) Operador Aggregate, el operador agrupa por IP y una vez se deslizan las ventanas se genera un evento por IP con el esquema: <*userTS*:long, IP:String, *sessionId*:String, *startTS*:long, *counter*:entero>. El campo *startTS* es el valor del campo *userTS* del primer evento de la ventana y el campo counter muestra el número de tuplas que se almacenaban la ventana hasta que se ha deslizado. 4) Operador Map, añade un campo de tipo long llamado *endTS*, la marca de tiempo del sistema en ese momento. Las tuplas tras este operador Map tienen el siguiente esquema: <*userTS*:long, IP:String, *sessionId*:String, *startTS*:long, *endTS*: long, *counter*:entero>. 5) Operador SocketDataSink, envía las tuplas a través de un socket a un cliente que recibe las tuplas procesados.

Al desplegar la consulta está queda dividida en dos sub-consultas: la Sub-Consulta1 que tiene el primer operador Map y la Sub-Consulta2 que consta de los operadores Aggregate y Map. El balanceador el operador DataSource va a ser de tipo round-robin y el balanceador de las instancias de la Sub-Consulta 1 es de tipo *GroupKey* ya que el operador Aggregate se configura de tal manera que agrupe las tuplas por IP.

El objetivo es mostrar el correcto funcionamiento y los tiempos de ejecución de la migración, scale up y scale-down. Además se va a mostrar como escala el UPM-CEP aumentando el número de instancias IM en una máquina (Bullion y XEON) y aumentando el número de máquinas (XEON).

5.5.1. Tolerancia a fallos: Replicación Activa

Durante la evaluación del sistema de tolerancia a fallos mediante replicación activa, se va a analizar el tiempo que tarda el sistema en reconfigurarse ante un fallo, así como el rendimiento del mismo cuando se están procesando los datos sin el sistema de tolerancia a fallos y con el sistemas de tolerancia a fallos activado. Para ello, se ha medido el tiempo que tardan en procesarse las tuplas (latencia) y el número de tuplas procesadas al segundo (throuhgput) en cada uno de los tres escenarios.

La evaluación se ha llevado acabo en el clúster Bullion, en el que los servicios se han configurado de la siguiente manera (figura 5.9): En un nodo diferente a los dos nodos Bullion (Nodo Inyector) están los clientes que envían y reciben las tuplas respectivamente (cliente1 y cliente2). En el nodo 1, el nodo NUMA 0 tiene los procesos orchestrator, metricserver y ZooKeeper, cada uno de ellos asignado a un core físico. El nodo NUMA 1 tiene tres instanceManagers (IM1, IM2 e IM3) cada uno de ellos asociado a un core físico y con 32 GB de RAM. Y en el nodo 2, el nodo NUMA 1 tiene un instanceManager (IM4) asociado a un core físico y con 32 GB de RAM. La consulta FixedTimeWindow ha sido desplegada en el clúster como se muestra en la figura 5.9. En los IM1 e IM2 se encuentran el DataSource y DataSink respectivamente y en los IM2 e IM4 se ha desplegado Sub-Consulta1-1 junto con Sub-Consulta2-1 y Sub-Consulta1-2 junto con Sub-Consulta2-2, respectivamente.

Se han realizado las evaluaciones de los tres escenarios presentados utilizando cuatro tamaños distintos de ventana (5, 10, 15 y 30 segundos) y cuatro cargas diferentes con cada tamaño de ventana (10.000, 20.000, 30.000 y 40.000 tuplas por segundo). Cada punto de la evaluación se ha repetido tres veces mostrando la media en las gráficas. Los resultados obtenidos con los tamaños de ventana 5, 10, 15 y 30 segundos se muestran en las figuras 5.10, 5.11, 5.12 y 5.13, respectivamente. A los 5 minutos de la ejecución se simula que falla el IM2 haciendo que el sistema tenga que configurarse para comenzar enviar las tuplas sin comprobación.

El throughput producido, como máximo es igual al número de IPs generadas, en este caso son 65.536 IPs. De esta manera, cuando se genera un número de tuplas su-

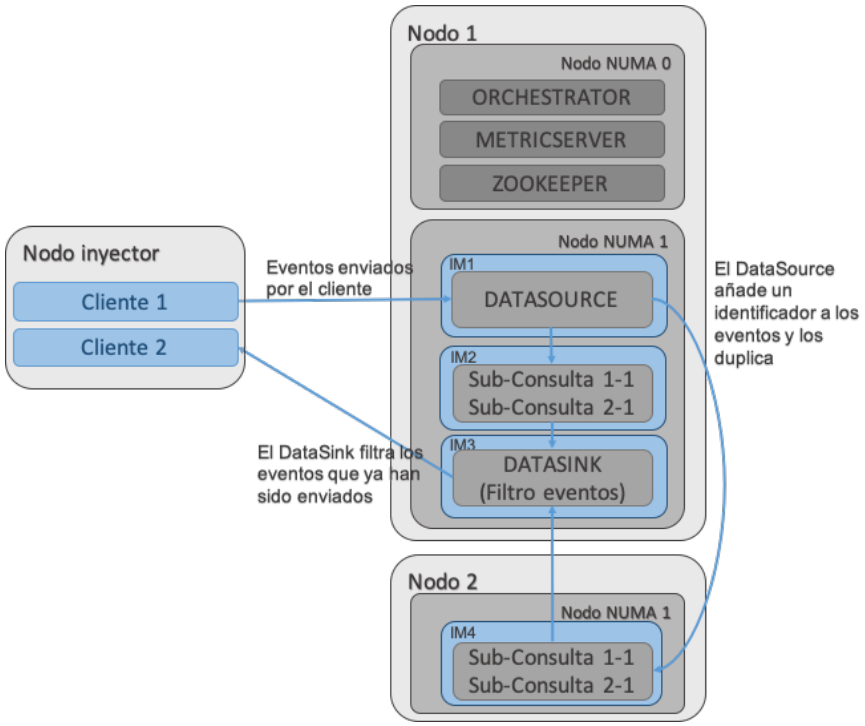


Figura 5.9: Consulta Fixed Time Window con Repliación Activa

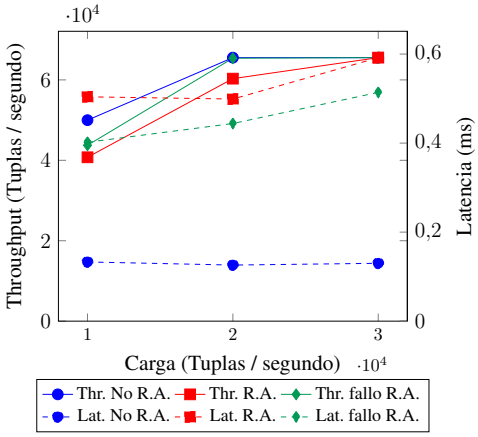


Figura 5.10: Replicación Activa Hi-Bench en Bullion - 5s

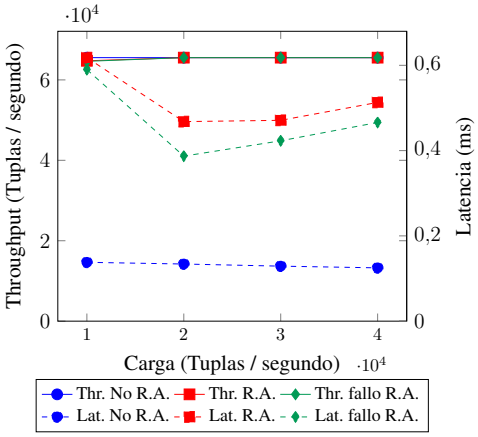


Figura 5.11: Replicación Activa Hi-Bench en Bullion - 10s

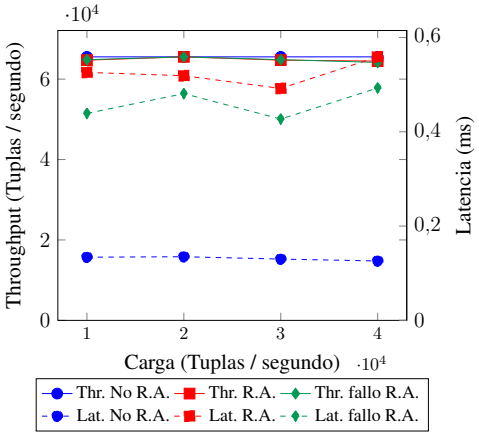


Figura 5.12: Replicación Activa Hi-Bench en Bullion - 15s

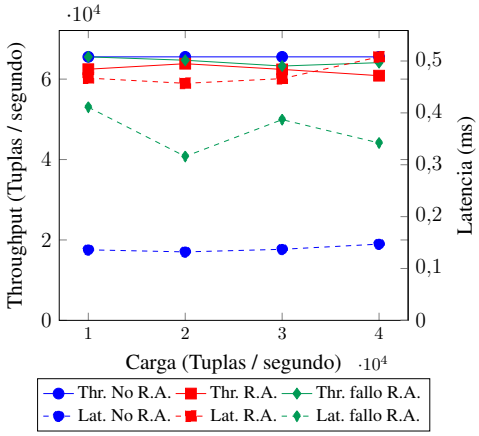


Figura 5.13: Replicación Activa Hi-Bench en Bullion -30s

ficiente durante los segundos del tamaño de la ventana, el operador Aggregate llega a tener tantas ventanas como IPs. Este efecto puede observarse en las figuras 5.11, 5.12 y 5.13, donde con cualquiera de las cargas se llegan a producir todas las IPs posibles. En cambio en la figura 5.10 donde se muestran los resultados con ventanas de 5 segundos, con la carga de 10.000 eventos al segundo se llegan a almacenar 50.000 tuplas, 15.536 menos que el máximo número de IPs.

El principal impacto del sistema de tolerancia a fallos se detecta en la latencia. La latencia, cuando la ejecución de la consulta es sin tolerancia a fallos, permanece estable entorno a 0,13 milisegundos en todas la ejecuciones (línea azul punteada). Cuando la ejecución de la consulta se realiza con tolerancia a fallos, tanto antes como después del fallo la latencia es superior, entorno a 0,51 milisegundos cuando están las dos réplicas en funcionamiento (línea roja punteada) y alrededor de 0,43 milisegundos cuando ha fallado una de las réplicas (línea verde punteada). Estos resultados se deben a que cuando están las dos réplicas el operador DataSink tiene que ir comprobando por cada una de las tuplas que se van a enviar al cliente, si ya ha sido enviada o no, lo cual supone una penalización de unos 0,38 milisegundos por evento. El tiempo medio que tarda en reconfigurarse el UPM-CEP tras el fallo es de 1,271 segundos. La latencia tras el fallo, sigue siendo superior a la obtenida sin tolerancia a fallos, esto se debe a: 1) el operador DataSink tiene que cambiar de forma de funcionamiento y 2) los eventos tienen que

ser enviados desde la máquina 1, procesados por la máquina 2 y de vuelta procesados por el operador DataSink en la máquina 1, lo que hace que la latencia sea superior a la ejecución sin tolerancia a fallos siendo aproximadamente 0,3 milisegundos superior.

Aunque la activación del sistema de tolerancia a fallos basado en replicación activa tenga una penalización de entorno a 0,45 milisegundos de latencia extra en el procesamiento de los eventos, esta es inferior a la diferencia mostrada por Flink. En la evaluación realiza sobre Flink con el método *checkpointing* se ha mostrado que supone una latencia extra de 2 segundos en la ejecución con el *checkpointing* activado. Además, UPM-CEP no se detiene en ningún momento a pesar de que haya ocurrido un fallo y una de las réplicas haya dejado de estar disponible. De esta manera, el sistema se configura sin que el cliente se percate del fallo ocurrido y sin obstaculizar el procesamiento de las tuplas que llegan al sistema.

5.5.2. Migración

Durante la evaluación de rendimiento del proceso de migración se va a medir el tiempo que tarda en mover una sub-consulta con estado desde un IM ejecutándose en un nodo hasta otro IM en otro nodo del clúster. Con ello se va a medir además, el tiempo que tarda en transferirse el estado, el número de ventanas y tuplas que se han transferidos y el tiempo que tarda en enviarse las tuplas que se han almacenado en el buffer del balanceador durante el proceso de transferencia de estado junto con el número de tuplas. Para realizar la evaluación se ha desplegado la consulta *Fixed Time Window* utilizando los tamaños de ventana de 5, 10, 15 y 30 segundos. La evaluación se ha llevado acabo en el clúster Bullion y AMD. Con cada despliegue se han realizado varias ejecuciones aumentando el tamaño de la carga desde las 10.000 tuplas por segundo hasta las 40.000 tuplas por segundo en el clúster Bullion y desde las 10.000 hasta las 80.000 tuplas por segundo en el clúster AMD.

En el clúster Bullion los servicios se han configurado de la siguiente manera (figura 5.14): En un nodo diferente a los dos nodos Bullion (Nodo Inyector) están los clientes que envían y reciben las tuplas respectivamente (cliente1 y cliente2). En el

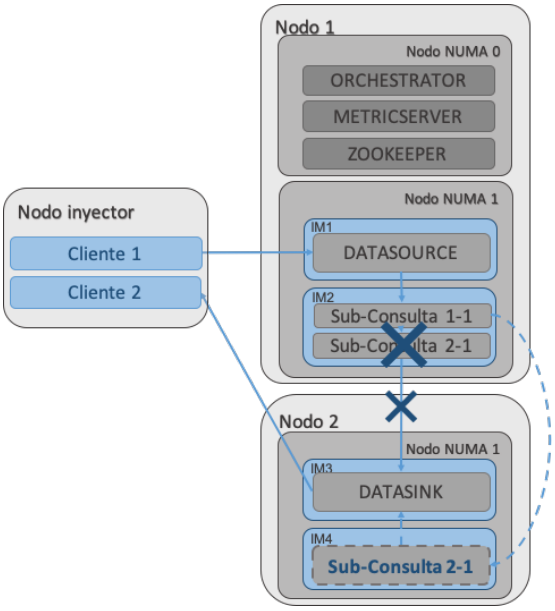


Figura 5.14: Migración sub-consulta 2 en clúster Bullion Sequana

nodo 1, el nodo NUMA 0 tiene los procesos orchestrator, metricserver y ZooKeeper cada uno de ellos asignado a un core físico. El nodo NUMA 1 tiene dos instanceManagers (IM1 e IM2) cada uno de ellos asociado a un core físico y con 32 GB de RAM. Y en el nodo 2 el nodo NUMA 1 de nuevo dos instanceManagers (IM3 e IM4) asociado a un core físico y con 32 GB de RAM. La consulta *Fixed Time Window* ha sido desplegada en el clúster como se muestra en la figura 5.14. En los IM1 e IM3 se encuentran el DataSource y DataSink respectivamente y en el IM2 se ha desplegado Sub-Consulta1-1 junto con Sub-Consulta2-1. Tras un tiempo el orchestrator decide migrar la Sub-Consulta2-1 desde el IM2 al IM4 dejando la Sub-Consulta1-1 sola en el IM2.

Durante la evaluación realizada en el clúster Bullion no se obtuvo las métricas que indican el número de tuplas almacenadas en el buffer ni el tiempo que tarda el balanceador en enviarlas tras terminar el proceso de transferencia de estado y además solo se pudo realizar una única ejecución de cada una de las evaluaciones. Por otro lado, se ha obtenido el tiempo del proceso de migración (figura 5.15), el tiempo del proceso de transferencia de estado (figura 5.16) y el número de ventanas y tuplas enviadas durante

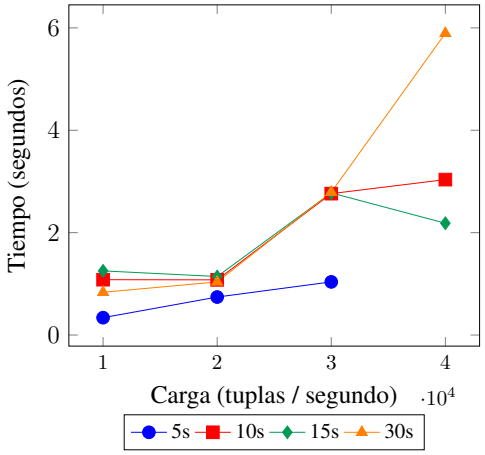


Figura 5.15: Tiempo Migración - Bullion

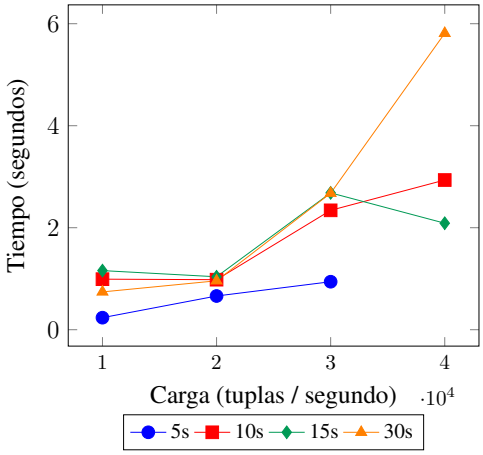


Figura 5.16: Tiempo transferencia del estado - Bullion

el proceso de transferencia de estado (figuras 5.17 y 5.18, respectivamente). En todas las figuras el eje x muestra la carga (tuplas/segundo), en las figuras 5.15 y 5.16 el eje, representa el tiempo en segundos que tarda en ejecutarse la migración y la transferencia de estado, mientras que en las figuras 5.17 y 5.18 representa el número de ventanas y tuplas, respectivamente.

La mayor parte del tiempo de migración se destina a la transferencia del estado, observando las figuras 5.15 y 5.16 estas son semejantes. Por ejemplo en la ejecución con tamaño de ventana de 30 segundos y carga de 40.000 tuplas por segundo el tiempo de migración es 5,894 segundos y el tiempo de transferencia de estado es 5,812 segundos, es decir el 98 % del tiempo de migración es utilizado por el proceso de transferencia de estado. El punto en el que más diferencia se observa entre ambos tiempos es en la ejecución con tamaño de ventanas de 10 segundos y carga de 30.000 tuplas al segundos (15 %) donde el tiempo de migración es de 2,767 segundos y el tiempo de transferencia de estado es de 2,341 segundos. Esta diferencia puede deberse al número de tuplas que han sido almacenadas durante el proceso de transferencia de estado y el tiempo que ha tardado el balanceador en enviarlas. De media en todas las ejecuciones el tiempo de transferencia supone un 91 % del tiempo de migración.

Por otro lado el tiempo que tarda la transferencia de estado tiene relación directa

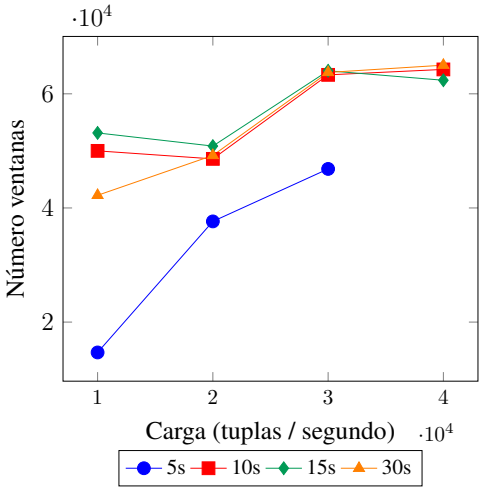


Figura 5.17: Número de ventanas transferidas - Bullion

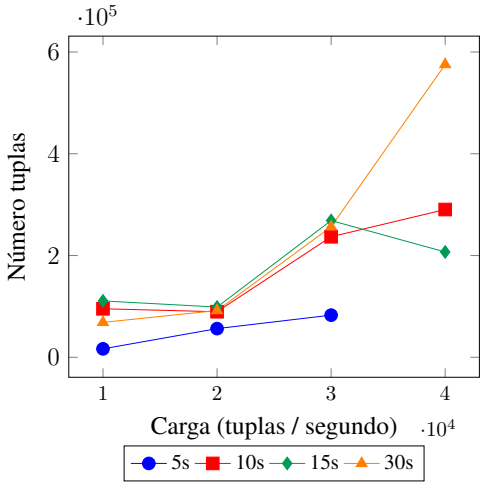


Figura 5.18: Número de tuplas transferidas - Bullion

con el número de ventanas y tuplas que son enviadas. Dependiendo del momento en el que se ha ejecutado el proceso de migración de la sub-consulta, el número de ventanas con tuplas almacenadas varía ya que puede darse el caso de que en ese momento se hayan desplazado alguna de las ventanas. Durante esta evaluación el máximo número de ventanas que pueden tener tuplas en el momento de la transferencia de estado es 65.536 y esto es debido al número de IPs que se están generando en el cliente. Por ello contra más grande es el tamaño de la ventana y mayor sea la carga, la cantidad de ventanas que hay que transferir aumenta. Por ejemplo en la evaluación con el tamaño de ventana de 5 segundos el número de ventanas a transferir con la carga de 10000 tuplas al segundo es de 14.680, mientras que con las cargas de 20.000 y 30.000 tuplas al segundo este se incrementa hasta los 37.643 y 46.838 ventanas, respectivamente. De todas maneras lo que realmente tiene un gran impacto en el tiempo de transferencia de estado es el número de tuplas. Por ejemplo en la evaluación con tamaño de ventana de 30 segundos y carga de 40000 tuplas por segundo, que se corresponde con el tiempo más alto en la transferencia de estado se han migrado 575.328 tuplas. Mientras que en la evaluación con el mismo tamaño de ventana y carga de 10000 tuplas al segundo se han enviado 68.617 tuplas por segundo dando lugar a un tiempo de transferencia de estado de 0,741 segundos frente a los 5,812 segundos de la evaluación con 40.000

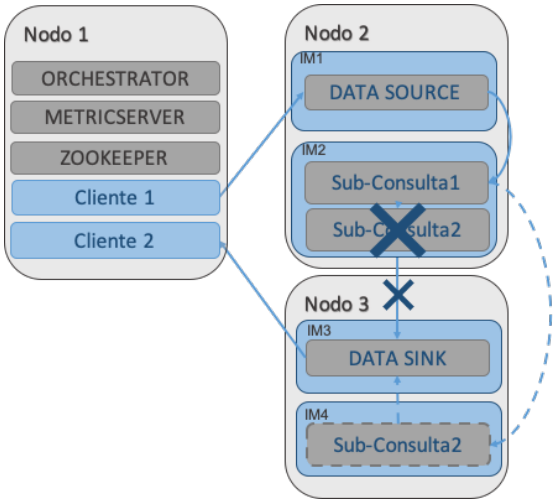


Figura 5.19: Migración sub-consulta 2 en clúster AMD

tuplas por segundo.

Para la evaluación en el clúster AMD se han utilizado tres nodos en los que los servicios han sido desplegados tal y como muestra la figura 5.19. El nodo 1 tiene los servicios orchestrator, metricserver, ZooKeeper y los clientes cliente1 y cliente2 en envían y reciben tuplas respectivamente. En los nodos nodo 2 y nodo 3 se ha deplegado dos IMs en cada uno de ellos (IM1, IM2, IM3 e IM4), asignando a cada uno de ellos 1 core físico y 4096 MB de RAM. Se ha deplegado la consulta *Fixed Time Window* de tal manera que el IM1 tiene en ejecución el operador DataSource, el IM2 tiene las sub-consultas Sub-Consulta1-1 y Sub-Consulta2-1, el IM3 tiene en ejecución el operador DataSink y al inicio de la ejecución el IM4 se encuentra libre. Tras un tiempo de ejecución el orchestrator decide migrar la Sub-Consulta2-1 al IM4, de tal manera que el IM2 solo va a tener en ejecución la Sub-Consulta1-1.

Durante la evaluación se han mantenido los tamaños de ventana, las cargas utilizadas van desde las 10.000 tuplas al segundo hasta 80.000 tuplas al segundo. Cada una de estas ejecuciones ha sido realizada en tres ocasiones mostrando para queda una de las métricas la media de las ejecuciones. Las figuras 5.20, 5.21 y 5.25 muestran el tiempo de ejecución de la migración, la transferencia de estado y el envío de las tuplas almacenadas durante la transferencia en el buffer del balanceador. Mientras las figu-

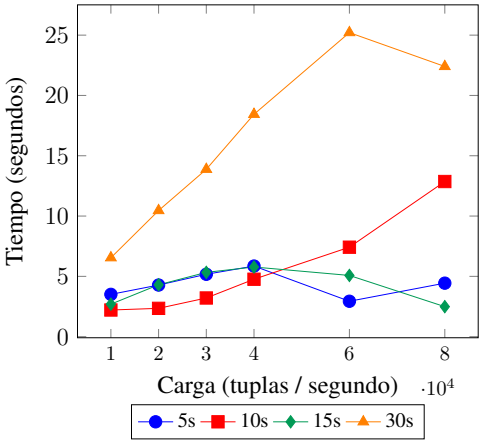


Figura 5.20: Tiempo Migración - AMD

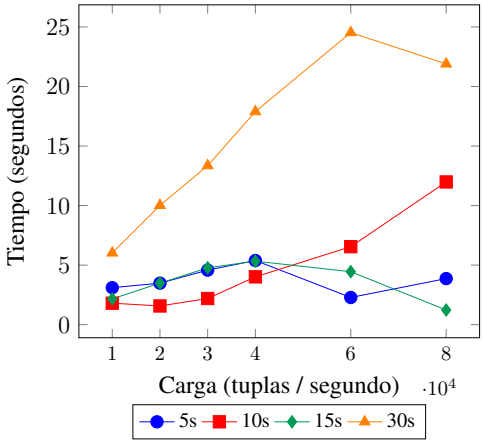


Figura 5.21: Tiempo transferencia del estado - AMD

ras 5.22, 5.23 y 5.24 representan el número de ventanas y tuplas enviadas durante la transferencia de estado y el número de tuplas que se han almacenado en el buffer del balanceador durante la transferencia del estado.

Observando las figuras 5.20 y 5.21 se puede observar que la mayor parte de tiempo de la migración se debe a la transferencia de estado. Un 86 % de media del tiempo de la migración es la transferencia de estado. Por ejemplo, para la evaluación con tamaño de ventana de 10 segundos y una carga de 40.000 tuplas al segundo, el tiempo de la migración es 4,762 segundos y el tiempo de la transferencia de estado es de 4,015 segundos (84 %). El resto del tiempo restante es el que lleva la creación de la instancia en el nuevo IM y el envío de la tuplas almacenadas durante la transferencia del estado. Siguiendo con el mismo ejemplo, el tiempo que ha tardado en enviar las 44.664 tuplas del buffer tras la transferencia del estado ha sido 0,451 segundos (9 %).

El tiempo que tarda en transferirse el estado está directamente relacionado con el número de tuplas que hay que enviar a la nueva localización de la instancia. Por ejemplo, el menor tiempo registrado en la transferencia del estado es con la configuración de ventana de 10 segundos y 20.000 tuplas al segundo de carga, 1,570 segundos donde se han enviado 19.868 tuplas pertenecientes a 15.518 ventanas. Por otro lado, con la configuración de 30 segundos de tamaño de ventana y una carga de 60.000 tuplas al

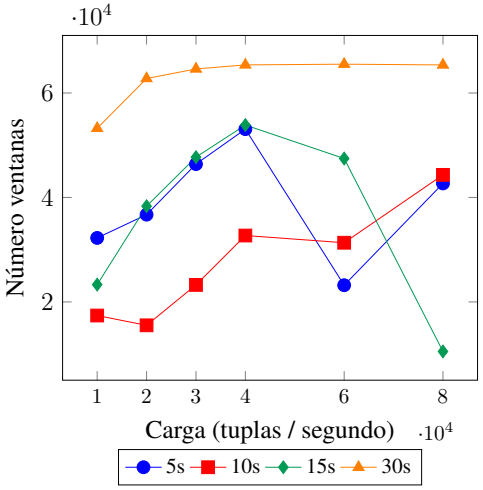


Figura 5.22: Número de ventanas transferidas - AMD

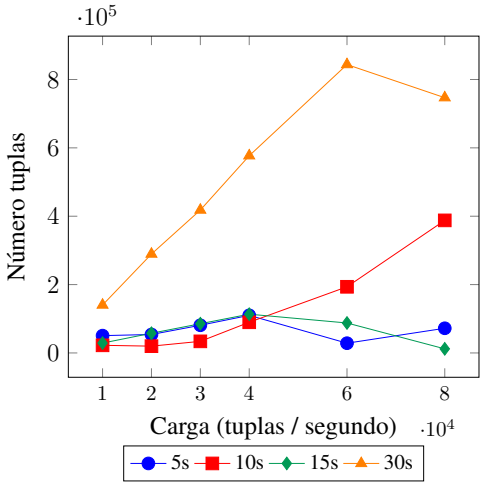


Figura 5.23: Número de tuplas transferidas - AMD

segundo se han enviado 843.737 tuplas pertenecientes a 65.516 ventanas durante la transferencia de estado, tardando 24,527 segundos en terminar la transferencia.

En cuanto a relación entre el tiempo que tardan en enviarse las tuplas que han sido almacenadas en el buffer y el tiempo total de la migración, este valor representa un aproximadamente un 9 % del total. En ninguna de las evaluaciones el tiempo supera el segundo y se llegan a enviar hasta 49.513 tuplas en 0,56 segundos en la evaluación con tamaño de ventana 10 segundos y 60.000 tuplas al segundo de carga. La ejecución en la que menos tuplas se han almacenado durante la transferencia del estado es con tamaño de ventana de 5 segundos y carga de 10.000 tuplas al segundo donde se han enviado en 0,112 segundos las 15.064 tuplas.

En las figuras 5.26 y 5.27, se puede ver en tiempo de ejecución el impacto que tiene la migración en la evaluación con ventana de 5 segundos y la carga de 80.000 tuplas al segundo. En un primer momento las sub-consultas Sub-Consulta1-1 y Sub-Consulta2-1 están ejecutándose en el mismo IM. Tras un tiempo la Sub-Consulta2-1 es migrada a otro IM dejando la Sub-Consulta1-1 sola en el IM inicial. En la figura 5.26 se muestra la evolución en el tiempo de la carga (línea verde, oculta tras la naranja), throughput (línea amarilla) y latencia (línea azul) de la Sub-Consulta1-1, durante los primeros 4 minutos la Sub-Consulta1-1 es capaz de procesar como máximo 58.000

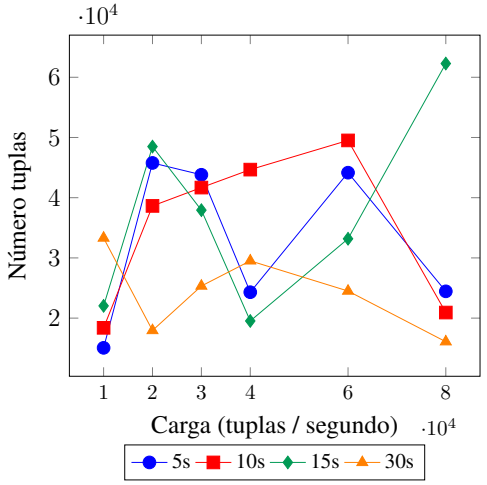


Figura 5.24: Número de tuplas almacenados en el buffer - AMD

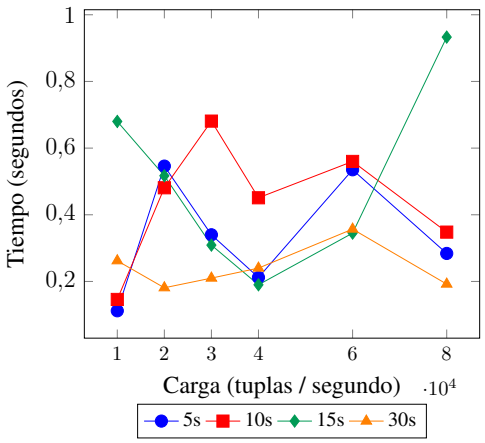


Figura 5.25: Tiempo envío tuplas almacenados en el buffer - AMD

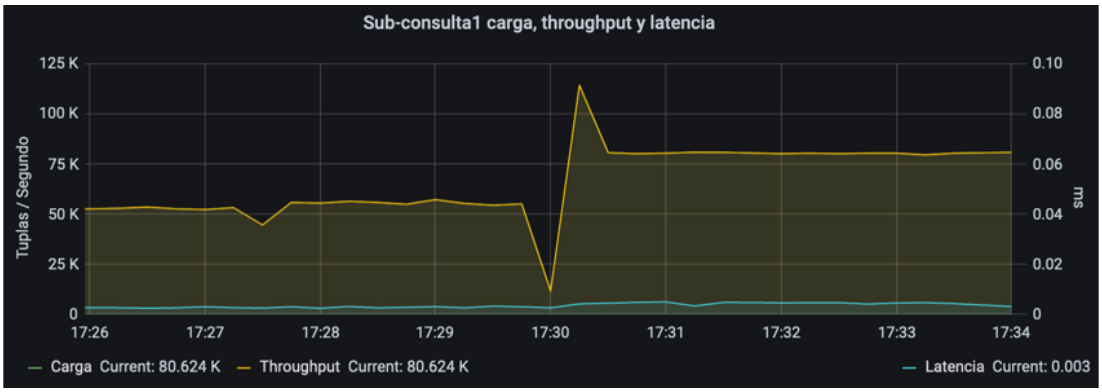


Figura 5.26: Migración: carga, throughput y latencia Sub-Consulta1-1

tuplas al segundo, tras realizar el migración de la Sub-Consulta2-1 la carga que procesa la Sub-Consulta1-1 llega a alcanzar las 80.000 tuplas al segundo que el cliente le está enviando. En la figura 5.27, muestra la carga (línea verde,naranja tras la migración), throughput (línea amarilla, roja tras la migración) y latencia (línea azul y azul oscuro tras la migración) de la Sub-Consulta2-1. Se puede observar que la carga procesada mientras la Sub-Consulta2-1 se encuentra en el mismo IM que la Sub-Consulta1-1 es de 58.000 tuplas al segundo aproximadamente y que al migrar la Sub-Consulta2-1 al nuevo IM esta sube a 80.000 tuplas al segundo. Aumentando de esta manera un 27 % la carga que la consulta es capaz de procesar.

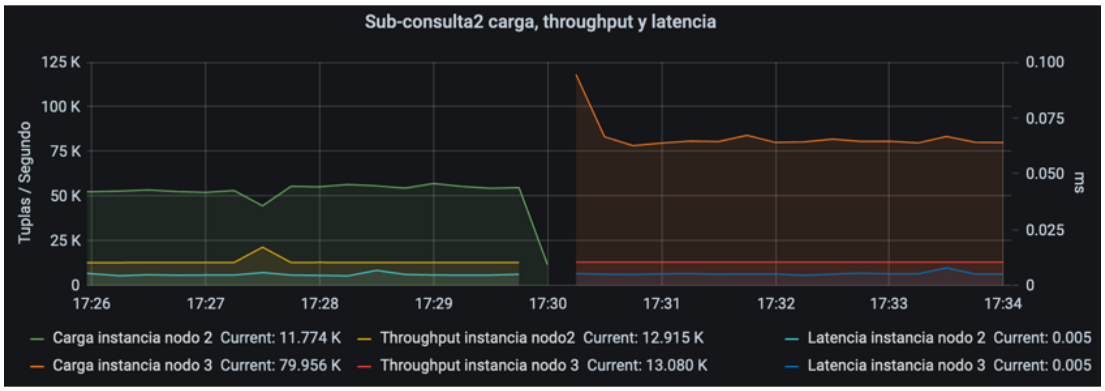


Figura 5.27: Migración: carga, throughput y latencia Sub-Consulta2-1

5.5.3. *scale-up*

La evaluación del proceso scale-up se ha realizado en el clúster AMD con el objetivo de evaluar el tiempo del proceso de escalado, el tiempo del proceso de transferencia de estado junto con el número de ventanas y número de tuplas que se transfieren a la nueva instancia y el número de tuplas que se han almacenado en el buffer del balanceador durante el proceso de transferencia del estado y el tiempo que ha tardado el balanceador en enviarlas a la instancia correspondiente una vez a terminado la transferencia del estado. Para realizar esta evaluación se han considerados dos escenarios: 1) peor caso, cuando la instancia de la sub-consulta a escalar es la única instancia desplegada (figura 5.28), 2) mejor caso: cuando la instancia de la sub-consulta a escalar no es la única instancia desplegada (figura 5.29). En ambos casos se han utilizado 2 nodos, en el nodo 1 están los servicios orchestrator, metricServer y ZooKeeper junto con el cliente que genera las tuplas y el cliente que los recibe (cliente 1 y cliente 2). En el peor escenario (figura 5.28), el nodo 2 tiene 5 IMs cada uno de ellos ha sido asignado a un core del nodo NUMA 0 o 1, ya que en un nodo NUMA pueden llegar a lanzarse 4 IMs, con 4096 MB de RAM. La consulta *Fixed Time Window* se ha desplegado como muestra la figura 5.28, donde los IM1 e IM2 tiene el DataSource y DataSink, respectivamente. El IM3 tiene una instancia de la Sub-Consulta1-1 y la Sub-Consulta2 tiene inicialmente una instancia desplegada en el IM4. Tras un tiempo de ejecución la instancia 1 de la Sub-Consulta2, se escala y se despliega en el IM5 la instancia Sub-

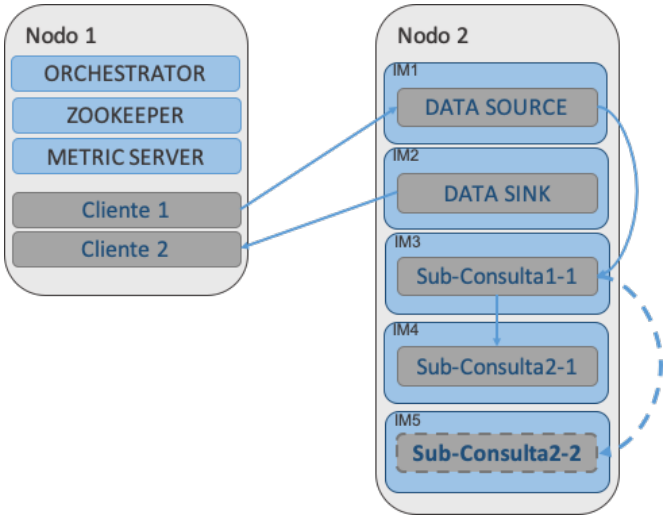


Figura 5.28: Peor escenario: Pasos *scale-up* sub-consulta 2-1

Consulta2-2 la cual va a tener la mitad de las ventanas de la Sub-Consulta2-1. Durante este proceso al no haber más instancias desplegadas, la Sub-Consulta2 deja de procesar tuplas hasta que se vuelve a establecer el flujo de datos del balanceador del stream de entrada a las dos instancias.

En el mejor escenario (figura 5.29), el nodo 2 tiene 6 IMs configurados con la misma configuración que en el peor escenario La consulta *Fixed Time Window* se ha desplegado como muestra la figura 5.29, donde los IM1 e IM2 tiene el DataSource y DataSink, respectivamente. El IM3 tiene una instancia de la Sub-Consulta1-1 y la Sub-Consulta2 tiene inicialmente dos instancias desplegadas en los IMs IM4 e IM5, un tercio de las tuplas son porcesadas por el instancia Sub-Consulta2-1 mientras que la instancia Sub-Consulta2-2 está procesando 2/3 de las tuplas que se procesan en la Sub-Consulta1. Tras un tiempo de ejecución la instancia 2 de la Sub-Consulta2, se escala y se despliega en el IM6 la instancia Sub-Consulta2-3 la cual va a tener la mitad de las ventanas de la Sub-Consulta2-2, de esta manera cada una de las instancias va a procesar un tercio de la carga. Durante el proceso de escalado la instancia Sub-Consulta2-1 va a seguir procesando las tuplas que le corresponden mientras que la Sub-Consulta2-2 va a estar detenida hasta que termine el proceso.

Para realizar la evaluación se han mantenido los tamaños de las ventanas de eva-

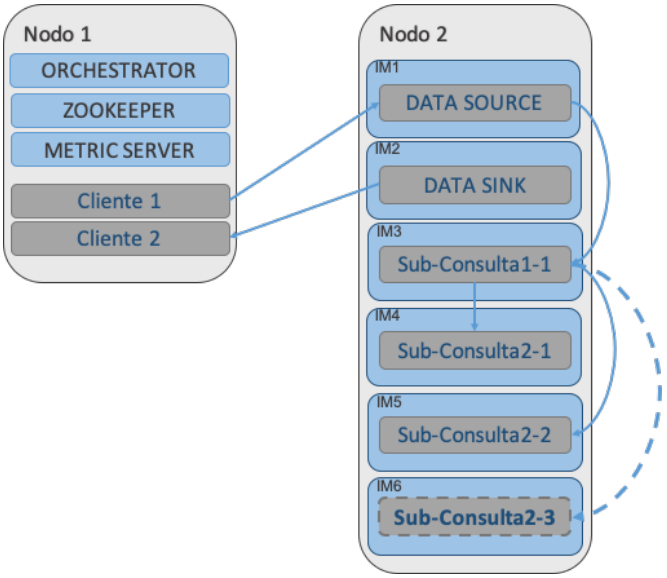


Figura 5.29: Mejor escenario: Pasos *scale-up* sub-consulta 2-2

luaciones anteriores y se han incrementado las cargas con 100.000, 110.000, 120.000, 130.000, 140.000 y 160.000 tuplas al segundo para la evaluación del peor escenario y para la evaluación del mejor escenario se han incrementado hasta 180.000 y 200.000 tuplas al segundo. Cada una de las evaluaciones se ha realizado tres veces y se muestran los resultados medios obtenidos. Las figuras 5.30, 5.31 , 5.38 y 5.39 muestran el tiempo de scale-up y el tiempo del proceso de transferencia de estado en las evaluaciones de ambos escenarios. Las figuras 5.32, 5.33, 5.40 y 5.41 muestran el número de ventanas y tuplas enviadas durante el proceso de transferencia de estado y las figuras 5.34, 5.35, 5.42 y 5.43 muestran las tuplas almacenadas en el buffer durante la transferencia y el tiempo que ha tardado el balanceador en enviarlas una vez se ha enviado el estado.

El tiempo de transferencia de estado (figura 5.31) en el peor escenario supone de media el 64 % del tiempo total del proceso scale-up (figura 5.30). Por ejemplo en la evaluación con tamaño de ventana de 15 segundos y 160.000 tuplas al segundo de carga, el tiempo de scale-up es 19,172 segundos mientras que el tiempo de transferencia del estado es de 11,217 segundos, lo que supone un 59 %. Durante este tiempo se han enviado 281.945 tuplas las cuales estaban almacenadas en 33.051 ventanas.

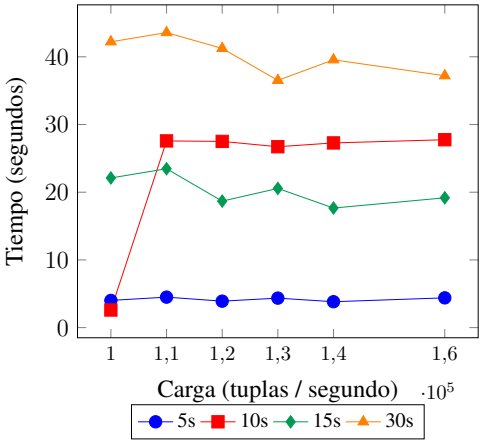


Figura 5.30: Scale-up peor caso: Tiempo scale-up

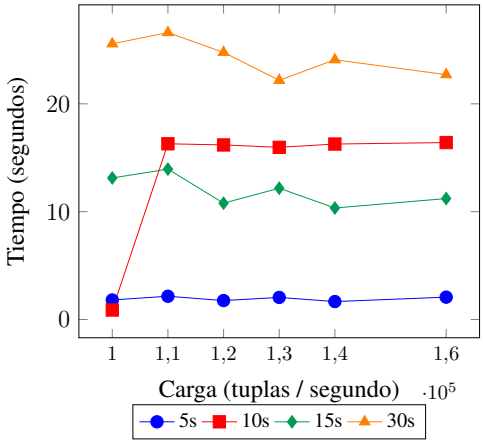


Figura 5.31: Scale-up peor caso: Tiempo transferencia del estado

El número de ventanas (figura 5.32) llega a estabilizarse en el máximo número de ventanas que pueden transferirse, 33.056 ventanas en las evaluaciones con tamaño de ventana igual o superior a 10 segundos. Este valor viene establecido por el número de IPs generadas por el cliente (65.536), al inicio de la ejecución la instancia Sub-Consulta2-1 está almacenando 65.536 ventanas (IPs). Al escalar la instancia Sub-Consulta2-1, la mitad de las ventanas que está sirviendo (33.056) se envían a la nueva instancia Sub-Consulta2-2. En cuanto al número de tuplas transferidas (figura 5.33), está directamente relacionado con el tiempo que tarda en finaliza la transferencia de estado. Por ejemplo en la evaluación con tamaño de ventana 30 segundos la transferencia de estado tarda aproximadamente 24 segundos y el número de tuplas enviadas es de 733.701 de media. Mientras que con tamaño de ventana de 5 segundos, el tiempo de transferencia es aproximadamente de 2 segundos y se envían 19.270 tuplas de media.

Finalmente observando la cantidad de tuplas almacenadas durante la transferencia de estado (figura 5.34), está directamente relacionado con el tiempo que tarda en transferirse el estado. Contra más tiempo dura la transferencia más tuplas se almacenan el buffer y por consiguiente más tiempo tarda el balanceador en enviar las tuplas almacenadas (figura 5.35). Este proceso tarda de media 8,6 segundos lo que supone un 41 % del total del tiempo del proceso scale-up. Por ejemplo en la evaluación con tamaño de ventana de 10 segundos y carga de 140.000 tuplas al segundo se envían 1.342.279

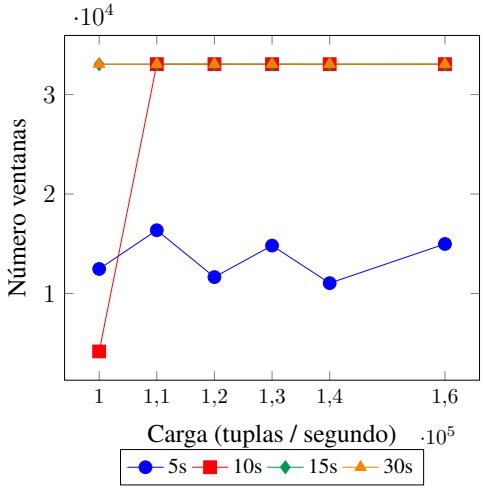


Figura 5.32: Scale-up peor caso: Número de ventanas transferidas

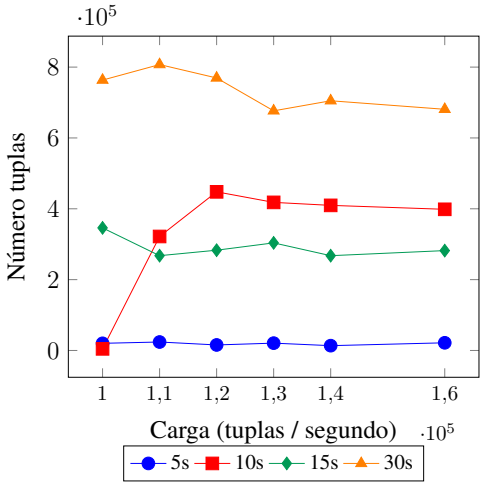


Figura 5.33: Scale-up peor caso: Número de tuplas tranferidas

tuplas en 10,657 segundos. En la evaluación que más tiempo tarda en realizarse la transferencia de estado (30 segundos tamaño de ventana y 110.000 tuplas al segundo de carga) se han enviado 2.213.511 tuplas en 16,635 segundos.

Las figuras 5.36 y 5.37 muestran la carga (tuplas/segundo), throughput (tuplas/-segundo) y latencia (milisegundos) en tiempo de ejecución de las sub-consultas Sub-Consulta1 y Sub-Consulta2 respectivamente. Durante esta evaluación se ha desplegado la consultas *Fixed Time Window* con un tamaño de ventana de 5 segundos y se ha enviado una carga de 130.000 tuplas por segundo. Durante los primeros minutos de la evaluación la consulta ha sido desplegada de tal manera que hay una instancia de la Sub-Consulta1 y una instancia de la Sub-Consulta2. Tras un tiempo la instancia de la Sub-Consulta2-1 es escalada y a partir de ese momento la carga inicial de la Sub-Consulta2-1 se distribuye entre ella y la nueva sub-consulta Sub-Consulta2-2.

En la figura 5.36 se muestra en la líneas verde y amarilla la carga y el throuhgput de la instancia Sub-Consulta1-1 donde se observa un valor de 100.000 tuplas al segundo durante los primeros 4 minutos de la evaluación y 130.000 tuplas al segundo durante los siguientes 4 minutos. En el minuto 16:31 de la evaluación se observa un descenso tanto de la carga como de throughput (aproximadamente 55.000 tuplas al segundo) y una subida de la latencia (línea azúl) para seguido aumentar la carga hasta las 130.000

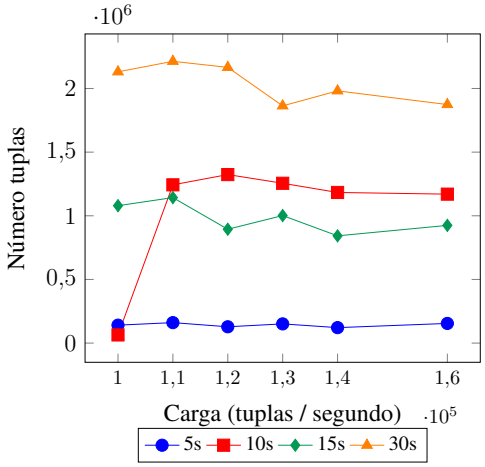


Figura 5.34: Scale-up peor caso: Número de tuplas almacenados en el buffer

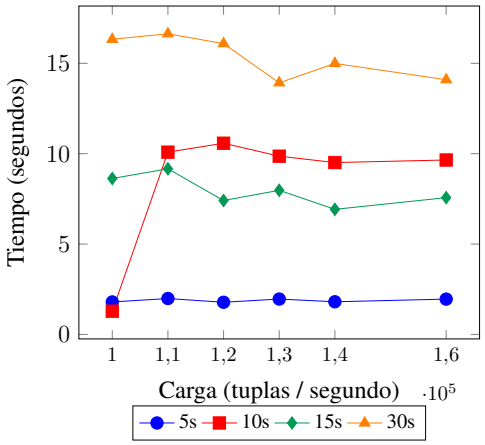


Figura 5.35: Scale-up peor caso: Tiempo envío tuplas almacenados en el buffer

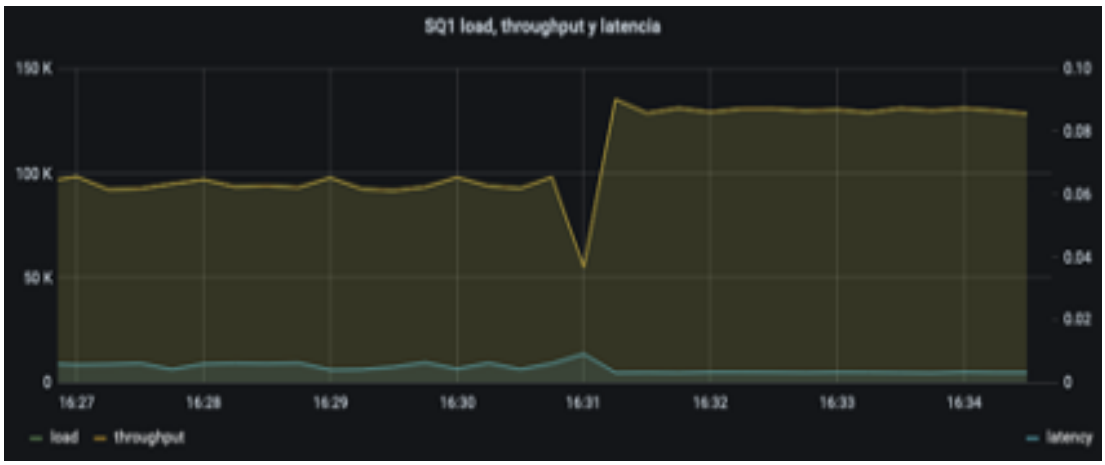


Figura 5.36: scale-up: cargar, throughput y latencia Sub-Consulta1-1

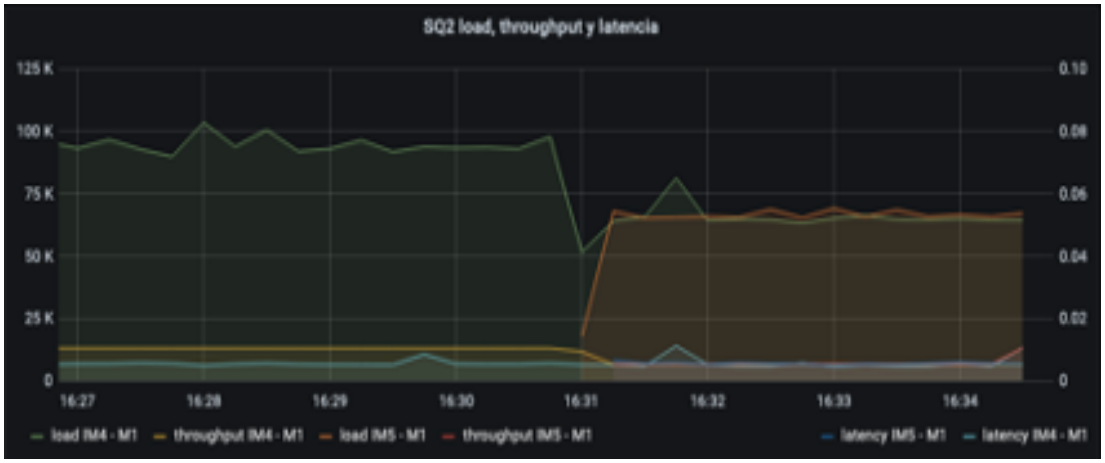


Figura 5.37: scale-up: cargar, throughput y latencia Sub-Consulta2

tuplas al segundo. Este efecto se produce por la ejecución del proceso scale-up de la Sub-Consulta2-1. Tras haber finalizado el scale-up la latencia desciende y permanece mucho más estable y tanto la carga como el throughput se mantienen en la carga que se está generando.

La figura 5.37 muestra la carga y el throuhgput de las instancias Sub-Consulta2-1 (líneas verde y amarillo) y Sub-Consulta2-2 (líneas naranja y rojo) a demás de la latencia de cada una de ellas (líneas azul y azul oscuro). Se observa que inicialmente la carga de 130.000 tuplas al segundo no puede ser procesada por la Sub-Consulta2-1 ya que como mucho llega a procesar 100.000 tuplas al segundo. Al realizarse el scale-up de la Sub-Consulta2-1, la carga se reduce a 65.000 tuplas al segundo y la carga de la nueva instancia Sub-Consulta2-3 se establece al mismo valor (65.000 tuplas al segundo). Con este protocolo de aumento del número de instancias permite procesar una mayor cantidad de tuplas al segundos, distribuyendo la carga entre más instancias.

En la evaluación considerando el mejor caso (figura 5.29), el tiempo de transferencia de estado (figura 5.39) supone de media el 66 % del tiempo total del proceso scale-up (figura 5.38). Por ejemplo en la evaluación con tamaño de ventana de 30 segundos y 180.000 tuplas al segundo de carga, el tiempo de scale-up es 33,34 segundos mientras que el tiempo de transferencia del estado es de 22,59, lo que supone un 68 %. Durante este tiempo se han enviado 704.247 tuplas las cuales estaban almacenadas en

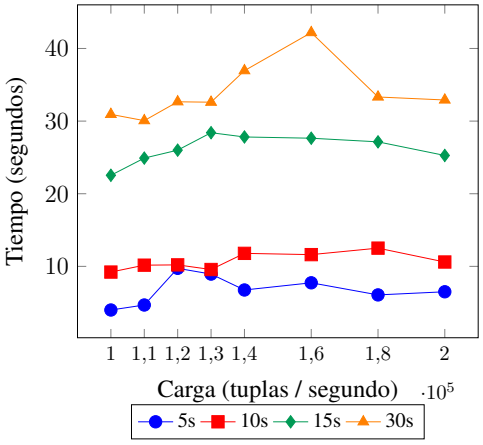


Figura 5.38: Scale-up mejor caso: Tiempo scale-up

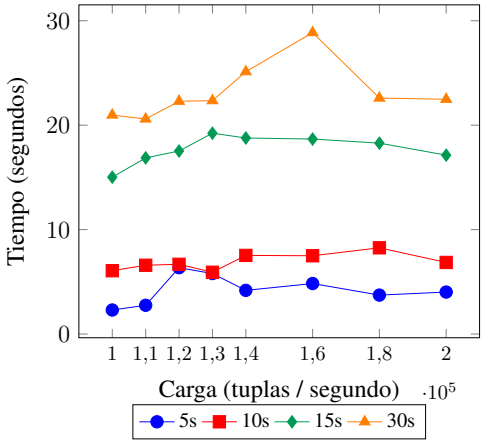


Figura 5.39: Scale-up mejor caso: Tiempo transferencia del estado

21.538 ventanas. En las evaluaciones en las que menos tiempo ha sido destinado a la transferencia de estado en comparación con el tiempo total de scale-up ha sido con tamaño de ventana de 5 segundos donde supone una media del 62 %, es resto del tiempo ha sido destinado al envío de la tuplas almacenadas durante la transferencia de estado y el registro y despliegue de la Sub-Consulta2-3.

El máximo número de ventanas a transferir es de 21.538 (figura 5.40), es decir un tercio de las IPs generadas (65536 IPs). Este valor se debe a que la Sub-Consulta2-2 está procesando dos tercios del número de ventanas y al ejecutar el proceso scale-up la mitas de las ventanas son transferidas a la nueva instancia Sub-Consulta2-3. En las evaluaciones con tamaños de ventana de 15 y 30 segundos, las 21.538 ventanas a transferir tenían tuplas almacenadas. El número de tuplas transferidas (figura 5.41), aumenta según va aumentando el tamaño de la ventana y la carga inyectada llegando a transferirse hasta 923.683 tuplas en 28,9 segundos en la evaluación con tamaño de ventana de 30 segundos y carga de 160.000 tuplas al segundo.

El tiempo de envío de las tuplas almacenadas en el buffer (figura 5.43) está relacionado con la cantidad de tuplas que se han almacenado (figura 5.42) y a su vez con el tiempo que ha tardado en transferirse el estado desde la instancia a escalar a la nueva. Por ejemplo, en la evaluación con tamaño de ventana de 30 segundos y carga de 160.000 tuplas al segundo se han almacenado 1.889.811 tuplas en el buffer durante

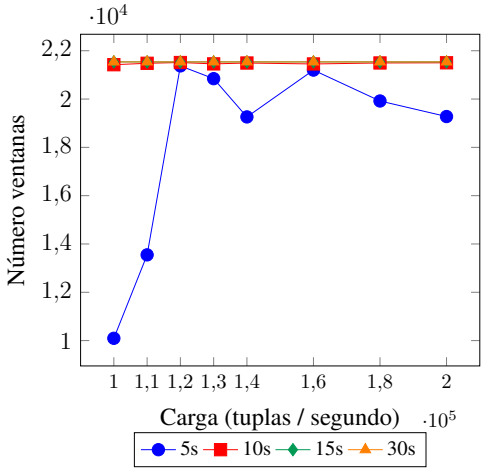


Figura 5.40: Scale-up mejor caso: Número de ventanas transferidas

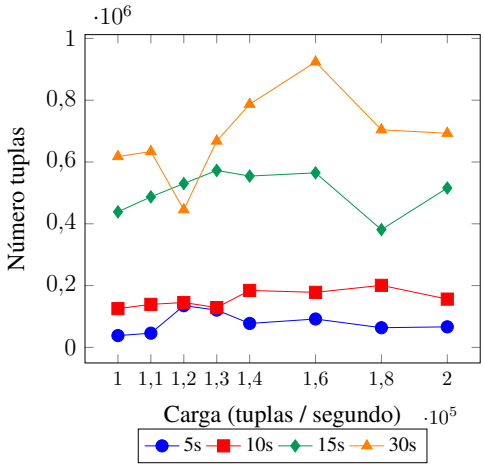


Figura 5.41: Scale-up mejor caso: Número de tuplas tranferidas

el proceso de transferencia del estado que han sido enviadas en 13,018 segundos. El tiempo que ha tardado en realizarse la transferencia de estado es de 28,866 segundos y se han enviado un total de 923.683 tuplas. Esto supone del total del tiempo de scale-up (42,186 segundos) el 68 % del tiempo se ha realizado la transferencia del estado y el 31 % ha sido utilizado para envío de las tuplas almacenadas en el buffer. De media en todas las evaluaciones, el tiempo de envío de las tuplas del buffer es de 32 % del tiempo total del proceso scale-up.

5.5.4. *scale-down*

El proceso de *scale-down* permite reducir el número de instancias para liberar recursos siempre y cuando las instancias que vayan a quedar en ejecución sean capaces de procesar la carga. Esta evaluación de rendimiento permite analizar el tiempo que tarda el sistema en configurarse al reducir el número de instancias de un Sub-Consulta. Para ello se ha obtenido las siguientes medidas, por un lado el tiempo que tarda en ejecutarse el proceso scale-down, el tiempo del proceso de transferencia de estado y el tiempo que tarda el balanceador del stream superior en enviar todas las tuplas que ha almacenado durante la transferencia de estado. Por otro lado se ha obtenido el número de ventanas y número de tuplas que se transfieren al resto de instancias de la sub-

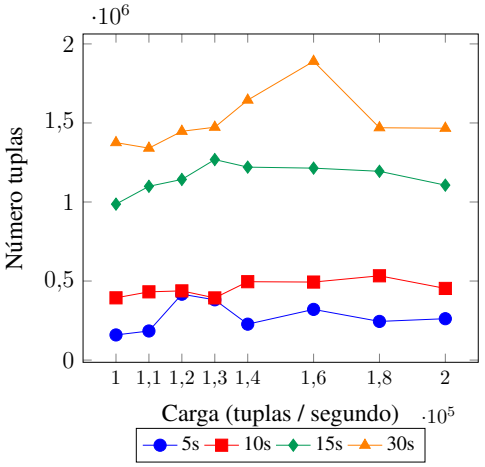


Figura 5.42: Scale-up mejor caso: Número de tuplas almacenados en el buffer

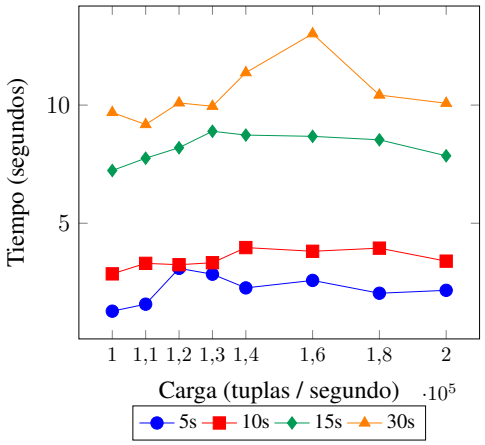


Figura 5.43: Scale-up mejor caso: Tiempo envío tuplas almacenados en el buffer

consulta que van a seguir en ejecución y el número de tuplas que se han almacenado en el buffer del balanceador. La evaluación se ha realizado el clúster AMD con dos nodos tal y como se muestra en la figura 5.44. El nodo 1 tiene los servicios orchestrator, metricServer y ZooKeeper junto con los clientes que generan las tuplas y las reciben. En el nodo 2 se han desplegado 5 IMs (IM1 - IM5) donde se ha desplegado la consulta *Fixed Time Window*. Los IMs IM1 e IM5 tiene los operadores DataSource y DataSink, respectivamente. El IM2 tiene una instancia de la Sub-Consulta1 y los IMs IM3 e IM4 tienen una instancia de la Sub-Consulta2 cada uno de ellos (Sub-Consulta2-1 y Sub-Consulta2-2).

El tamaño de las ventanas se mantiene con respecto al resto de evaluaciones (5, 10, 15 y 30 segundos) y las cargas son 10.000, 20.000, 30.000, 40.000, 60.000 y 80.000 tuplas al segundo. Cualquiera de las cargas indicadas puede ser procesada por una sola instancia de la Sub-Consulta 2. Cada una de las ejecuciones se ha realizado tres veces y los resultados mostrados se corresponde a la media. Tras un tiempo de ejecución se lanza el proceso scale-down sobre la instancia de la Sub-Consulta2-2, la instancia Sub-Consulta2-1 comienza a procesar las tuplas de las dos instancias.

El 71 % de media del tiempo del proceso de scale-down es destinado a la trans-

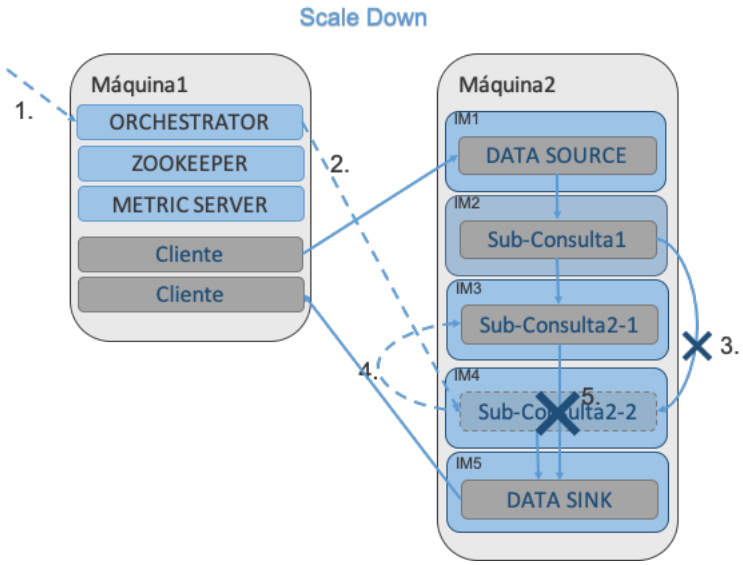


Figura 5.44: Pasos *scale-down* sub-consulta 2

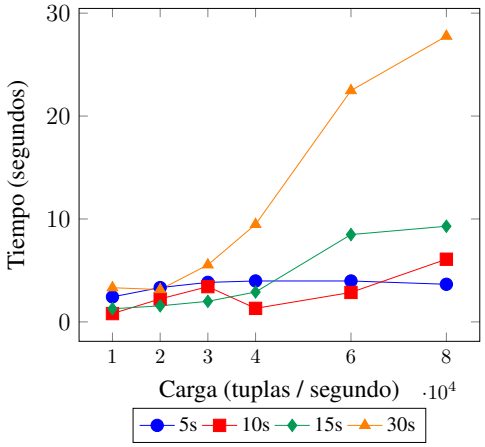


Figura 5.45: *scale-down* Sub-Consulta2 HiBench en AMD - Tiempo Migración

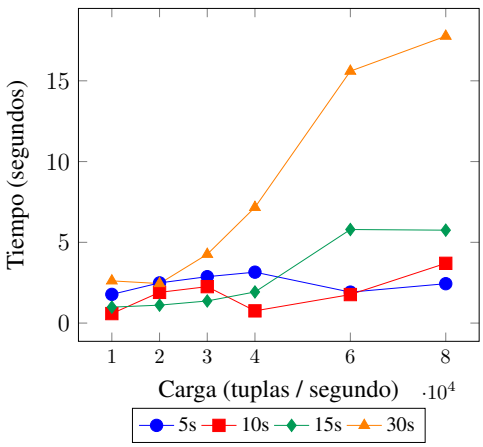


Figura 5.46: *scale-down* Sub-Consulta2 HiBench en AMD - Tiempo envío tuplas

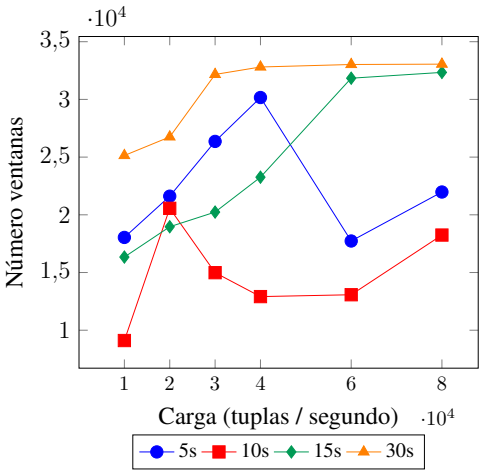


Figura 5.47: *scale-down* Sub-Consulta2 HiBench en AMD - Número de ventanas

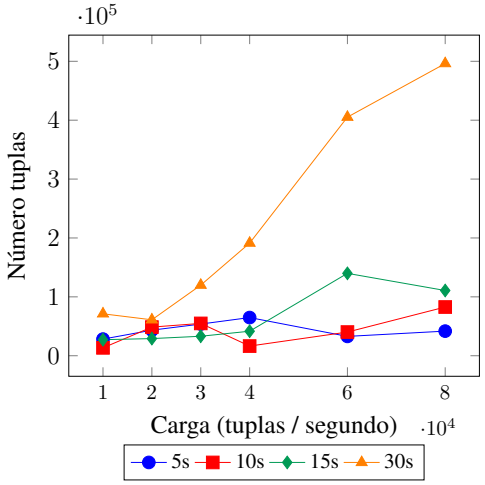


Figura 5.48: *scale-down* Sub-Consulta2 HiBench en AMD - Número de tuplas

ferencia de estado de la instancia a eliminar (Sub-Consulta2-2) a la instancia Sub-Consulta2-1. Por ejemplo, en la evaluación con tamaño de ventana de 25 segundos y carga de 30.000 tuplas al segundo, el tiempo de scale-down es de 1,999 segundos mientras que el tiempo de transferencia de estado es de 1,369 segundos lo que supone un 68 % del tiempo. Este hecho se repite en el resto de evaluaciones donde en el caso que menos tiempo se destina a la transferencia de estado en comparación con el tiempo total es con tamaño de ventana de 10 segundos y 80.000 tuplas al segundo de carga, 6,089 y 3,7 segundos lo que supone un 61 % del tiempo de scale-down.

En cuanto al número de ventanas ha ser enviadas a la Sub-Consulta2-1 se corresponde con la mitad de las IPs generadas por el cliente que son 65536, valor que se alcanza con los tamaños de ventana de 15 y 30 segundos y las cargas 80.000 tuplas al segundo. El número de ventanas que se envía depende del momento en el que se realice el proceso scale-down ya que puede haber ventanas que no tengan ninguna tupla que enviar. El número de tuplas enviadas tiene una relación directa con el tiempo que tarda en realizarse la transferencia de estado. Por ejemplo la evaluación que más tiempo destina en la transición del estado (17,764 segundos) envía 496.188 tuplas a la instancia Sub-Consulta2-1, con el tamaño de ventana de 30 segundos y 80.000 tuplas

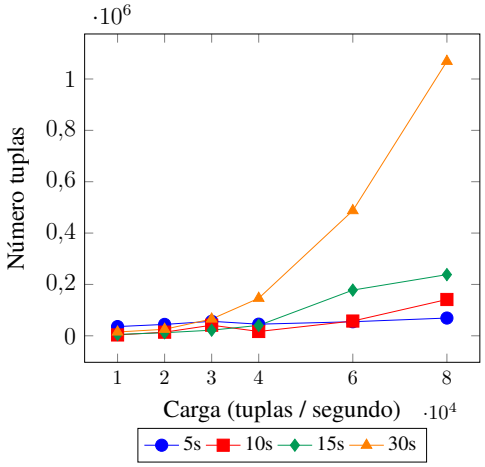


Figura 5.49: *scale-down* Sub-Consulta2 HiBench en AMD - Número de tuplas almacenadas durante el cambio

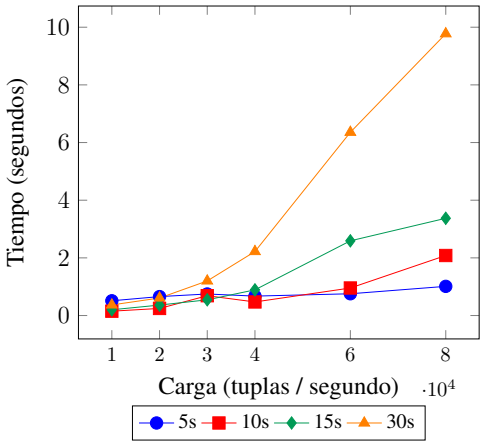


Figura 5.50: *scale-down* Sub-Consulta2 HiBench en AMD - Tiempo envío tuplas almacenadas durante el cambio

al segundo de carga.

El número de tuplas que se almacenan en el buffer del balanceador llega a alcanzar el millón de tuplas en la evaluación de 30 segundos de tamaño de ventana y 80.000 tuplas al segundo de carga, esta cantidad de tuplas se envía en 9,773 segundos. Por otro lado en la evaluación con tamaño de ventana 5 segundo y 10.000 tuplas al segundo se han enviado 3.286 tuplas en 0.151 segundos.

Las figuras 5.51, 5.52 muestran la evolución en el tiempo de la ejecución del proceso scale-down en las sub-consultas Sub-Consulta1 y Sub-Consulta2 de la consulta *Fixed Time Window* con tamaños de ventana 10 segundos y una carga de 80.000 tuplas al segundo. La carga utilizada es procesada en todo momento por la consulta tanto cuando hay dos instancias o una de la Sub-Consulta2. Por ello en la figura 5.51 se puede observar como la carga y el throuhgput permanecen estables en 80.000 tuplas al segundo durante todo el tiempo.

Por otro lado en la figura 5.52, se observa como durante los primero minutos de la ejecución la carga se distribuye entre las dos instancias de la Sub-Consulta2 (40.000 tuplas al segundo) y tras terminar el proceso scale-down en el que la instancia Sub-Consulta2-2 es eliminada del despliegue, la Sub-Consulta2-1 comienza a procesar la

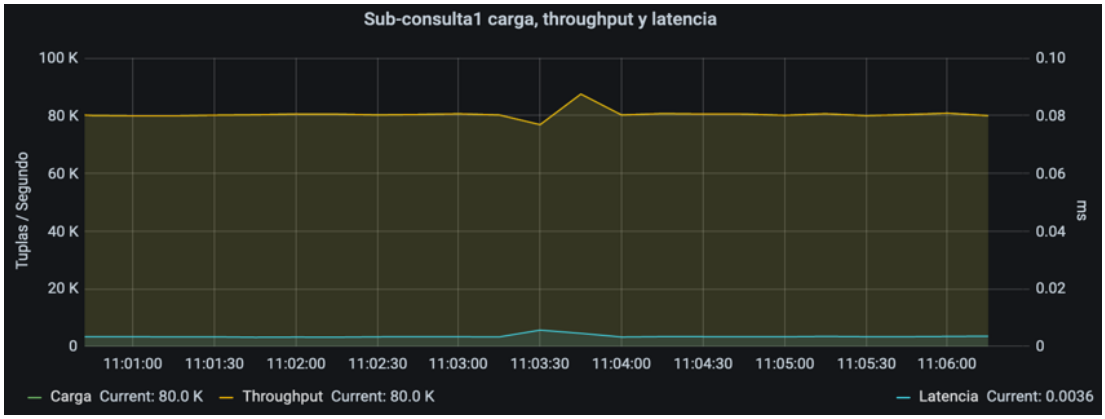


Figura 5.51: scale-down: cargar, throughput y latencia Sub-Consulta1-1

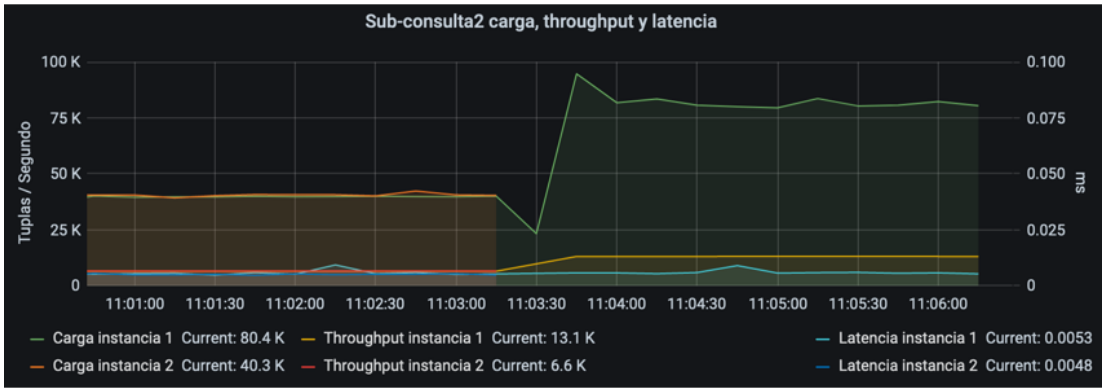


Figura 5.52: scale-down: cargar, throughput y latencia Sub-Consulta2-1 y Sub-Consulta2-2

carga total de 80.000 tuplas al segundo. También se puede observar que el Throuhgput de la Sub-Consulta2-1 se ve incrementado y eso es debido a que está enviando el doble de tuplas al operador DataSink. De esta manera se puede observar como el proceso scale-down permite liberar recursos del sistema cuando la carga puede ser procesada por menos recursos sin detener el procesamiento de tuplas en el resto de instancias.

5.5.5. Escalabilidad

Se ha realizado una evaluación de escalabilidad del UPM-CEP con el objetivo de demostrar que el rendimiento del sistema escala tanto en una única máquina como utilizando un clúster, asignando cada IM a un core físico aprovechando la arquitectura

NUMA de las máquinas. La siguiente evaluación se ha llevado acabo en un nodo Bullion y en el clúster XEON utilizando dos y 5 nodos. Para realizar esta evaluación se ha utilizado la consulta *Fixed Time Window* fijando el tamaño de ventana a 5 segundos y se ha ido aumentando el número de instancias de cada Sub-Consulta en cada ejecución.

En el nodo Bullion los servicios se han desplegado tal y como se muestra en la figura 5.53. En el nodo NUMA 0 se han desplegado los servicios orchestrator, metricserver y ZooKeeper, en el resto nodos NUMA pares (2, 4 y 6) se van a lanzar los clientes que van a enviar datos al CEP. Y en los nodos NUMA 1, 3, 5 y 7 se han desplegado hasta 60 InstanceManagers (IM1 - IM60), llegando a ocupar los recursos de la máquina. Para evitar que los operadores DataSource y DataSink supongan un cuello de botella durante la evaluación se han eliminado del despliegue de la consulta y son los clientes que envían tuplas los que van tener los balanceadores que van a enviar las tuplas directamente a las instancias de la Sub-Consulta1 y los clientes que reciben va a recibir las tuplas procesadas de las instancias de la Sub-Consultas2. De esta manera cada nodo NUMA impar va a tener hasta 15 IMs en ejecución tal y como se muestra la figura 5.54, cada IM ha sido asignado a un core físico del nodo NUMA y se le han asignado 32 GB de RAM. Por cada instancia de la Sub-Consulta2 hay dos instancias de la Sub-Consulta1 por lo que cada nodo NUMA impar va a tener 10 instancias de la Sub-Consulta1 y 5 instancias de la Sub-Consulta2. En cuando a los clientes cada uno de ellos va a producir una carga de 190.000 tuplas al segundo y va a mandar la carga a una instancia de la Sub-Consulta1, por lo tanto van a lanzarse un total de 40 clientes para enviar las tuplas y 1 cliente que recibirá las 65.536 tuplas que se van a producir (tantas como IPs se generan).

En la figura 5.55 se muestran los resultados, donde se fue incrementando el número de instancias de la consulta comenzando con un despliegue de 3 IMs, en la que hay dos instancias de la Sub-Consulta 1 y una instancia de la Sub-Consulta 2, hasta ocupar por completo la máquina con 60 IMs, con 40 instancias de la Sub-Consulta 1 y 20 instancias de la Sub-Consulta 2. La línea azul muestra la carga que se está enviando a la consulta en cada una de las ejecuciones comenzando con 380.000 tuplas al segundo para la evaluación con el despliegue más pequeño y terminando con 7,6 millones de

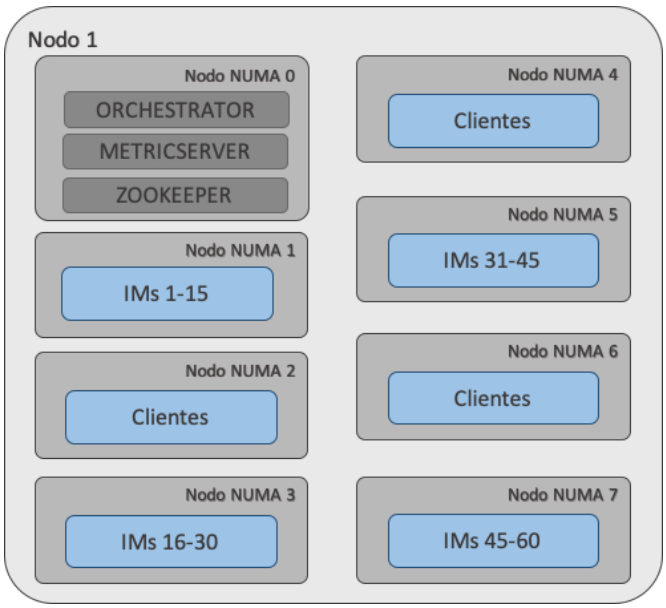


Figura 5.53: Distribución servicios escalabilidad Bullion

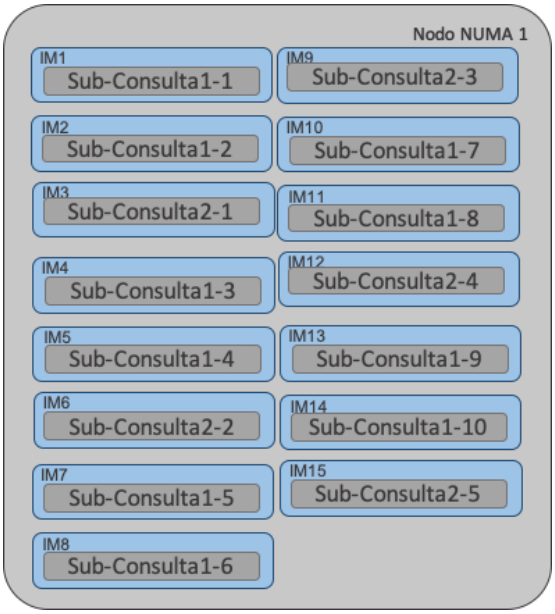


Figura 5.54: IMs y Sub-Consultas nodo NUMA 1 del Bullion

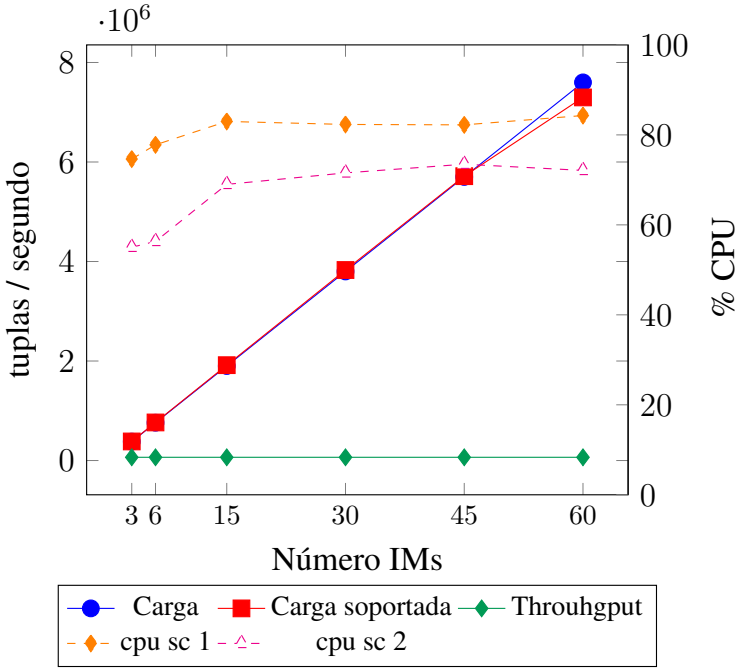


Figura 5.55: Escalabilidad Consulta HiBench en bullion

tuplas al segundo en la evaluación con 60 IMs. La línea roja muestra el número de tuplas al segundo que la consulta es capaz de procesar, en todo momento es igual a la carga generada por los clientes excepto en la evaluación con 60 IMs en la que se están utilizando todos los recursos de la máquina y donde desciende a 7,3 millones de tuplas al segundo, aproximadamente 300.000 tuplas al segundo menos. La línea verde muestra el número de tuplas que se producen de salida de la consulta que permanece estable en 65.536 tuplas al segundo debido a que ese el es número máximo de IPs que se generan. Y las líneas discontinuas muestran el consumo de CPU permaneciendo estable entorno al 83 % de media en los IMs con instancias de la Sub-Consulta 1 y el 75 % de media en los IMs con instancias de la Sub-Consulta 2. Esta diferencia se debe a que la Sub-Consulta 2 durante 4 segundos almacena las tuplas y a continuación procesa las ventanas. En cambio la Sub-Consulta 1 tiene un flujo continuo de tuplas a procesar, lo cual se traduce en un mayor consumo de CPU.

En la evaluación realizada en el clúster XEON se muestra por un lado la escalabilidad de un nodo y luego se van añadiendo nodos progresivamente hasta alcanzar 4

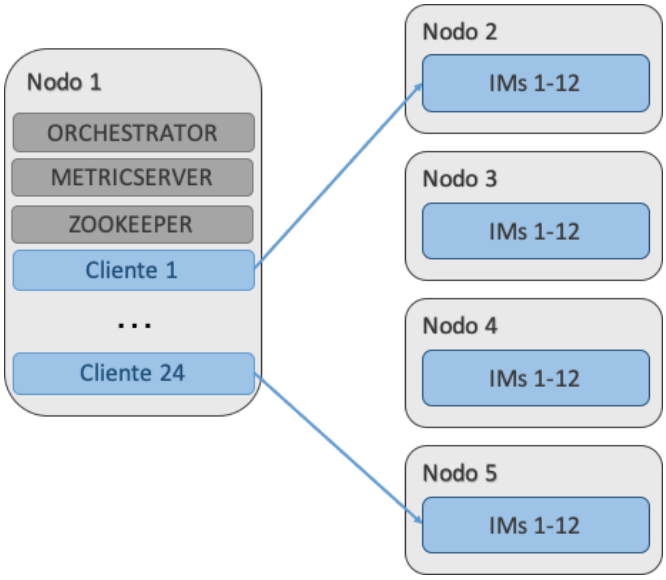


Figura 5.56: Distribución servicios escalabilidad Bullion

nodos con IMs. Los servicios en la evaluación con uno y cuatro nodos se distribuyen de la misma manera (figura 5.56. Uno de los XEON es utilizado para ejecutar los servicios orchestrator, metricserver y ZooKeeper a demás de tener los clientes y el resto van a tener hasta 12 IMs, llegando a tener hasta 72 IMs en total con los cuatro nodos. Cada uno de los IMs se han asignado a un core físico y se le ha asignado 10 GB de memoria RAM. La consulta *Fixed Time Window* se ha desplegado de la siguiente manera, por cada instancia de la Sub-Consulta2 hay dos instancias de la Sub-Consulta1, de tal manera que en un nodo XEON hay 8 instancias de las Sub-Consulta1 y 4 instancias de la Sub-Consulta2 (figura 5.57. Cada uno de los clientes va a generar una carga de 200.000 tuplas/al segundo que van a ser enviadas a dos instancias de la Sub-Consulta1, de esta manera en total se van a lanzar 24 clientes cuando se realice la evaluación con los 4 nodos.

Las figuras 5.58 y 5.59 muestran los resultados obtenidos en un XEON, donde se ha comenzado con 3 IMs, con dos instancias de la Sub-Consulta 1 y una instancia de la Sub-Consulta 2. Llegando a utilizar la máquina por completo con 12 IMs, con 8 instancias de la Sub-Consulta 1 y 4 instancias de la Sub-Consulta dos. Las cargas generada por las clientes ha sido desde 200.000 tuplas por segundo con un cliente y la

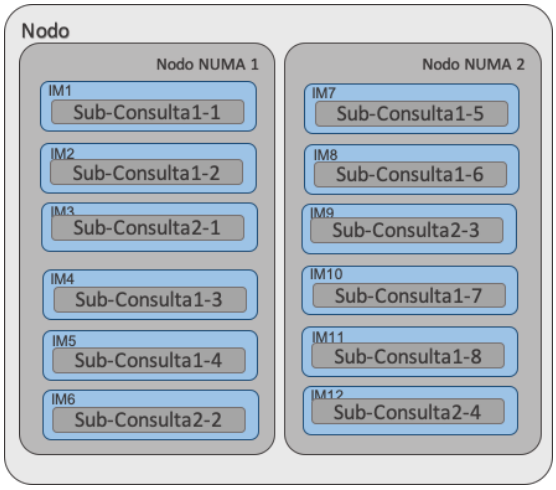


Figura 5.57: IMs y Sub-Consultas nodo NUMA 1 del Bullion

evaluación con 3 IMs, hasta las 800.000 tuplas por segundo con 4 clientes y utilizando toda la máquina. La carga generada por los clientes es procesada en todo momento excepto en el último punto en el que se utiliza toda la máquina y donde se llegan a procesar 776.854 tuplas al segundo, 23.146 tuplas al segundo menos. Esto se debe a que al estar ocupando toda la máquina parte de los recursos los consume el sistema operativo, lo que hace que el IM que se encuentra ubicado en el core que comparte con el sistema operativo no tenga la misma capacidad que el resto perjudicando así el rendimiento de la máquina. El consumo de CPU (figura 5.58) permanece estable en todas las ejecuciones, entorno a 80-75 % de media en los IMs que tienen la Sub-Consulta 1 y entre el 55-45 % de media en los IMs que tienen la Sub-Consulta 2. En cuanto al consumo de memoria (figura 5.59 línea punteada) se puede observar que va incrementando progresivamente según se van añadiendo nuevas instancias y se va aumentando la carga ya que el número de eventos que tienen que almacenarse en las ventanas es superior en cada ejecución.

Las figuras 5.60 y 5.61 muestran la escalabilidad aumentando el número de nodos XEON desde los resultados obtenidos con una máquina hasta los 4 nodos. Para escalar se ha utilizado la configuración de 9 IMs en un solo XEON ya que es la configuración más grande en un solo nodo procesa la carga que se envía, 600.000 tuplas al segundo.

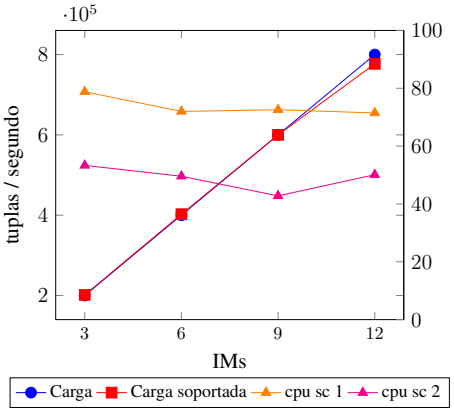


Figura 5.58: Escalabilidad Consulta HiBench en un XEON - Consumo CPU

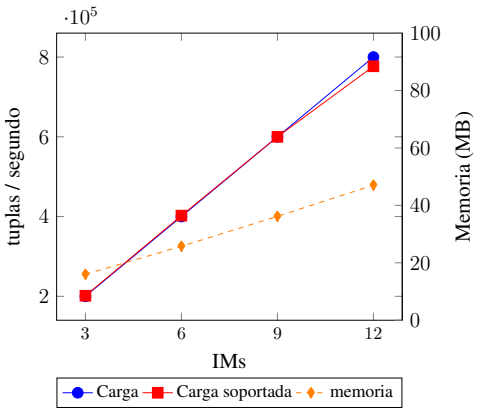


Figura 5.59: Escalabilidad Consulta HiBench en un XEON - Consumo memoria

Las máquinas se van añadiendo con la misma configuración que en un solo nodo, es decir, cuando hay dos nodos hay 18 IMs en ejecución enviándose 1,2 millones de tuplas al segundo, con tres nodos hay 27 IMs con una carga de 1,8 millones tuplas al segundo. Con cuatro nodos hay 36 IMs recibiendo una carga de 2,4 millones de tuplas al segundo. En todas las ejecuciones de puede observar que la carga es procesada por la consulta ya que es siempre igual a la carga que se envía. En cuando al consumo de CPU (figura 5.60) se encuentra entorno al 75 % de media en los IMs que tienen instancias la Sub-Consulta 1 y entre el 60-55 % de media en los IMs que tienen instancias la Sub-Consulta 2. La figura 5.61 muestra el consumo de memoria en cada ejecución comenzando con 36 GB en la evaluación con un nodo y llegando a consumir 151 GB en la evaluación con 4 nodos.

5.6. Conclusiones

En este capítulo se han presentado tres protocolos implementados durante el desarrollo de esta tesis que permiten otorgar al UPM-CEP las funcionalidades necesarias para que sea elástico y pueda adaptarse a las distintas cargas que recibe a lo largo del tiempo sin detener el procesamiento. Además se ha demostrar el rendimiento de cada uno de los protocolos y se ha realizado una demostración de que UPM-CEP es ca-

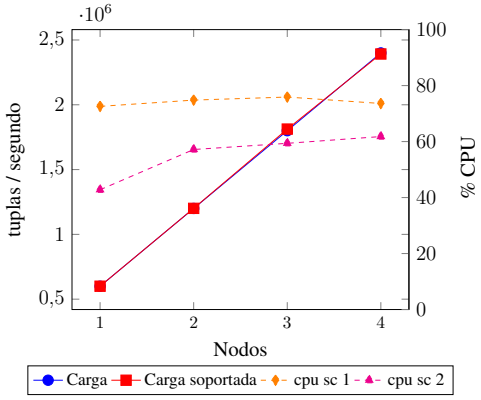


Figura 5.60: Escalabilidad Consulta HiBench en 4 XEON - Consumo CPU

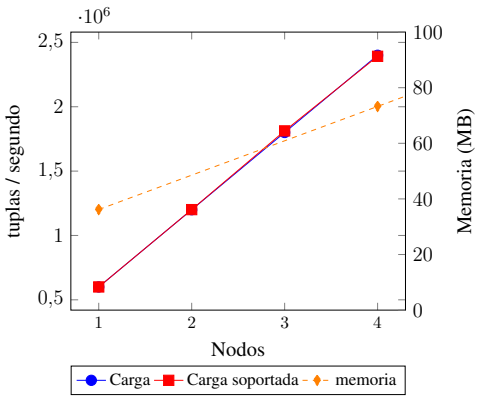


Figura 5.61: Escalabilidad Consulta HiBench en 4 XEON - Consumo memoria

paz de escalar tanto aumentado el número de servicios instanceManager dentro de un mismo nodo, como añadiendo nuevos nodos al sistema. Parte de este trabajo ha sido presentado en los congresos 9th International Conference on Cloud Computing and Services Science (CLOSER 2019) con el título Data Streaming for Appliances [52] y 8th International Conference on Data Science, Technology and Applications (DATA 2019) con el título Dynamic Data Streaming for an Appliance [53].

Además se ha mostrado un protocolo para dotar de tolerancia a fallos a uno de los componentes del UPM-CEP mediante un mecanismo de replicación activa y se ha mostrado una evaluación realizada al sistemas de tolerancia a fallos de Flink basado en checkpointing. Esta evaluación ha sido presentada en el congreso 24th International Conference on Parallel and Distributed Computing (Euro-Par 2018) con el título Cost of Fault-Tolerance on DataStream Processing [54]. Gracias a dicha evaluación se ha obtenido un punto de referencia a tener en cuenta para continuar con la implementación de otros sistemas de tolerancia a fallos en UPM-CEP.

Capítulo 6

Conclusiones y trabajo futuro

Durante esta tesis se han propuesto varios protocolos que permiten mejorar el rendimiento de dos tecnologías para el procesamiento y gestión de datos. Entre ellos, se ha desarrollado un conjunto de herramientas que permiten la carga de forma masiva y distribución de datos de manera uniforme entre los nodos de una base de datos NoSQL, HBase. Además se han implementado distintas técnicas para dotar a un sistema de procesamiento de datos en Streaming desarrollado en el Laboratio de Sistemas Distribuido de la Universidad Politécnica de Madrid, de elasticidad dinámica. A continuación se resumen las principales contribuciones realizadas en esta tesis de forma más detallada.

- **Carga masiva de datos desde ficheros CSV a HBase.** Tras analizar la herramienta que HBase proporciona para realizar la carga masiva de datos desde fichero a tablas (ImportTSV [14]), se observó que los ficheros tienen que tener una estructura determinada en la cual una de las columnas del fichero a carga tiene que ser la clave de las filas. Además la columna que contiene la clave de la fila no forma parte del conjunto de columnas de la tabla. HBase realiza las consultas teniendo en cuenta la clave, por lo que si se conocen con antelación las consultas que se realizan con mayor frecuencia se puede definir una clave que sea la concatenación de los campos más accedidos durante las consultas para asegurar un mejor rendimiento en el acceso a los datos unas vez estén cargados. En esta tesis se ha presentado, implementado y evaluado un algoritmo basado en la herramienta ImportTSV, el cual permite cargar de forma masiva datos en una

tabla previamente creada en HBase desde un fichero y generando al mismo tiempo las claves de las filas. De esta manera, todas las columnas del fichero forman parte de la tabla y las claves generadas permiten obtener un mejor rendimiento al realizar las consultas.

- **Creación de regiones con la misma cantidad de datos.** Tras crear una tabla en HBase esta se crea en un único nodo del sistema y se prepara para almacenar los datos en un única región. Conforme la región va creciendo, cuando alcanza un umbral establecido la región se divide en dos regiones con la misma cantidad de datos y ambas permanecen en el mismo nodo. Este proceso paraliza la carga de datos ya que HBase tiene que realizar la copia de los datos desde la región en la que se encuentran los datos a las dos nuevas regiones que se crean y posteriormente eliminar la región inicial. Para evitar este bloqueo del proceso de carga hay que crear las regiones y distribuirlas entre los nodos del sistema antes de comenzar con la carga. De esta manera el proceso de carga no se verá bloqueado en ningún momento y además se puede paralelizar la carga entre todos los nodos reduciendo de esta manera el tiempo de carga. A pesar de que HBase ofrece una serie de algoritmos que permiten crear regiones cuando la tabla está vacía, estos no son útiles cuando se desea realizar una distribución uniforme de los datos a cargar para conseguir el máximo paralelismo posible. Por ello se ha diseñado, implementado y evaluado un protocolo, que mediante el muestreo del fichero a cargar se obtienen las claves por las cuales se van a generar las regiones, crea tantas regiones como nodos del sistema y las distribuye entre los mismos para conseguir el máximo paralelismo posible durante el proceso de carga.

- **Distribución de recursos utilizando la arquitectura NUMA de las máquinas.** Con la aparición de las arquitecturas NUMA en la máquinas actuales se puede asociar procesos a cores y por consiguiente a secciones de memoria. Cuando un proceso no está asociado a ningún core en concreto, inicialmente se ejecuta en uno de los cores disponibles de la máquina y por consiguiente utiliza la memoria necesaria que se encuentra más accesible desde el core. Eventualmente, Linux

lanza un proceso que gestiona los procesos en ejecución para distribuir la carga de las CPUs por lo que los procesos en ejecución pueden ser desplazados desde el core de una CPU en concreto al core de otra CPU. Este proceso no mueve los datos almacenados en la memoria reservada al inicio de la ejecución del proceso por lo que los procesos tienen que acceder a las zonas de memoria que se encuentran en otra CPU. Para evitar esto, se puede asignar un proceso a un conjunto de cores desde el inicio de la ejecución y Linux no los moverá en ningún momento. Aprovechando esta propiedad de las máquinas con arquitectura NUMA se desarrollado, implementado y evaluado un algoritmo de distribución de los cores de una máquina entre los servicios que van a ser ejecutados, aprovechando el máximo número de recursos disponibles en la máquina. Este algoritmo puede ser utilizado para cualquier conjunto de servicios y cualquier tipo de arquitectura NUMA tal y como se muestra en esta tesis.

- **Distribución de los recursos asignados a YARN.** YARN utilizado durante el proceso de carga como gestor de recursos para MapReduce puede ser configurado de diversas maneras. En esta tesis se ha desarrollado un algoritmo que permite distribuir los recursos anteriormente asignados a YARN, mediante la herramienta de distribución de servicios en máquinas con arquitectura NUMA. La herramienta permite distribuir los cores virtuales y memoria RAM de tal manera que se creen tantos contenedores MapReduce como sean necesarios para paralelizar al máximo el proceso de carga.

- **Redistribución de la claves de las tablas HBase de manera uniforme.** Durante la vida de una base de datos distribuida, la distribución de las claves puede verse alterada debido modificaciones en el número de nodos disponibles en el sistema o a que alguna de las regiones de las tablas han recibido una mayor cantidad de datos. Estas modificaciones pueden afectar al rendimiento de las consultas cuando se realiza un acceso uniforme de los datos, debido a que unos nodos pueden tener más datos a los que acceder que otros haciendo que el tiempo de acceso

a los mismos sea superior en unos nodos que en otros. Para evitar este comportamiento, se ha implementado un algoritmo que permite volver a distribuir las claves entre los nodos disponibles del sistema de manera que cada uno de ellos termine sirviendo una cantidad aproximadamente igual de claves.

- **Tolerancia a Fallos mediante replicación activa en el UPM-CEP.** Con el objetivo de asegurar el correcto funcionamiento de las consultas en ejecución en el UPM-CEP en caso de que se produzca un fallo en las instancias del UPM-CEP, se ha implementado el sistema de tolerancia a fallos mediante replicación activa. De esta manera, las consultas son replicadas en distintos instanceManagers del sistema permitiendo que las mismas sigan procesando las tuplas a pesar de que se produzca un fallo en alguna de las réplicas.

- **Migración de sub-consultas entre distintos nodos del UPM-CEP.** En algunas ocasiones cuando la carga procesada por alguno de los instanceManagers se ve incrementada y en el mismo han desplegado un operador DataSource o DataSink y una o más instancias de sub-consultas o varias instancias de sub-consultas. Se ha implementado un proceso que permite mover una de las sub-consultas a otro instanceManager sin detener el procesamiento de las tuplas en el resto de instancias de la sub-consulta, aumentando de esta manera la carga procesada por la consulta.

- **Scale-up, scale-out: Modificación del despliegue de consultas del UPM-CEP para soportar una mayor carga.** Durante todo el tiempo de ejecución de las consultas, puede darse momentos en los que la carga que recibe el UPM-CEP se vea incrementada haciendo que los recursos utilizados por la consulta para procesar las tuplas recibidas no sean suficientes. En ese momento, UPM-CEP detecta que el consumo de CPU es muy alto o que la cantidad de memoria disponible es muy baja en alguno de los instanceManagers y comienza a desplegar nuevas instancias de la sub-consulta que esté en ejecución en el instanceMana-

ger saturado en otro instanceManager sin detener el procesamiento del resto de instancias y permitiendo de esta manera procesar la carga que está recibiendo.

- **Scale-down: Modificación del despliegue de consultas del UPM-CEP para ajustarse a una menor carga y liberar recursos.** De igual manera, si se reduce la carga que estaba siendo procesada por la consulta en el UPM-CEP y la carga actual puede ser procesada por menos instancias, se eliminan instancias de las sub-consultas permitiendo liberar recursos. Para conseguirlo se ha implementado un proceso que tras comprobar que los recursos de los instanceManager no están siendo utilizados, localiza la instancia a ser eliminada y redistribuye la carga que estaba siendo procesada por la instancia a eliminar entre el resto de instancias de las sub-consultas evitando de esta manera que se dejen de procesar tuplas durante el proceso de scale-down.

6.1. Trabajo Futuro

Tras la revisión de este trabajo se ha generado nuevas líneas de trabajo futuras detalladas a continuación:

- **HBase: Distribución de las claves teniendo en cuenta distintas distribuciones de acceso a las mismas.** El balanceador de carga ha sido implementado para distribuir las claves de manera uniforme entre los nodos del sistema teniendo en cuenta que las consultas que se realizan siguen una distribución uniforme de acceso a los datos. Como trabajo futuro se van a implementar otros algoritmos que permitan distribuir la carga de manera uniforme entre los nodos del sistemas teniendo en cuenta distintas distribuciones de los datos, como puede ser la distribución Zipfian.

- **UPM-CEP: Implementar sistemas de tolerancia a fallos.** Actualmente UPM-CEP tiene implementado la tolerancia a fallos mediante el algoritmo de replicación activa. Como trabajo futuro se van a implementar sistemas de tolerancia a

fallos como *backup replication*, en el que se lleva un registro de las tuplas procesadas hasta el momento del fallo y se vuelven a enviar las tuplas que no han podido ser procesadas por completo para asegurar que no se pierden tuplas.

- **UPM-CEP: Evaluación en redes de área extendida.** Las evaluaciones realizadas se han llevado acabo en clústers colocados en la misma área. Debido al aumento de las tecnologías IoT y a las características del UPM-CEP que le permiten ser desplegado en redes de área extendida, los siguientes pasos a realizar son: el despliegue del UPM-CEP en utilizando instanceManagers distribuidos en diferentes países utilizando nodos de AWS o Google y realizar pruebas de tolerancia a fallos, migración, scale-up, scale-out y scale-down.

Capítulo 7

Referencias

[1] SANJAY GHEMAWAT, HOWARD GOBIOFF, AND SHUN-TAK LEUNG. **The Google File System**. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. IX, IX, 1, 10, 11, 15, 17

[2] JEFFREY DEAN AND SANJAY GHEMAWAT. **MapReduce: Simplified Data Processing on Large Clusters**. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. IX, 2, 10, 19, 20, 22

[3] MIKE BURROWS. **The Chubby Lock Service for Loosely-coupled Distributed Systems**. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association. IX, 23, 25

[4] FAY CHANG, JEFFREY DEAN, SANJAY GHEMAWAT, WILSON C. HSIEH, DEBORAH A. WALLACH, MIKE BURROWS, TUSHAR CHANDRA, ANDREW FIKES, AND ROBERT E. GRUBER. **Bigtable: A Distributed Storage System for Structured Data**. *ACM Trans. Comput. Syst.*, **26**(2):4:1–4:26, June 2008. IX, IX, 2, 9, 25, 31, 32

[5] APACHE. **YARN Default XML file**. XV, 63, 65

[6] DANIEL J. ABADI, DON CARNEY, UGUR ÇETINTEMEL, MITCH CHERNIACK, CHRISTIAN CONVEY, SANGDON LEE, MICHAEL STONEBRAKER, NESIME TATBUL, AND STAN ZDONIK. **Aurora: A New Model and Architecture for Data Stream Management**. *The VLDB Journal*, **12**(2):120–139, August 2003. 2, 11

[7] DANIEL J. ABADI, YANIF AHMAD, MAGDALENA BALAZINSKA, UGUR ÇETINTEMEL, MITCH CHERNIACK, JEONG-HYON HWANG, WOLFGANG LINDNER, ANURAG MASKEY, ALEX RASIN, ESTHER RYVKINA, NESIME TATBUL, YING XING, AND STANLEY B. ZDONIK. **The Design of the Borealis Stream Processing Engine.** In *CIDR*, pages 277–289, 2005. 2, 11

7. REFERENCIAS

[8] APACHE. **Spark**. 2, 10

[9] CLOUDERA. **Cloudera**. 3

[10] CLOUDERA. **Hortonworks**. 3

[11] SAP. **SAP**. 3

[12] MAPR. **MapR**. 3

[13] APACHE. **Flink**. 3, 10, 140

[14] APACHE HBASE TEAM. **HBase Tools and Utilities. 130.11: ImportTsv**. 4, 55, 58, 189

[15] VINOD KUMAR VAVILAPALLI, ARUN C. MURTHY, CHRIS DOUGLAS, SHARAD AGARWAL, MAHADEV KONAR, ROBERT EVANS, THOMAS GRAVES, JASON LOWE, HITESH SHAH, SIDDHARTH SETH, BIKAS SAHA, CARLO CURINO, OWEN O'MALLEY, SANJAY RADIA, BENJAMIN REED, AND ERIC BALDESCHWIELER. **Apache Hadoop YARN: Yet Another Resource Negotiator**. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM. 4, 36

[16] V. GULISANO, R. JIMENEZ-PERIS, M. PATINO-MARTINEZ, AND P. VALDURIEZ. **StreamCloud: A Large Scale Data Streaming System**. In *2010 IEEE 30th International Conference on Distributed Computing Systems*, pages 126–137, 2010. 5, 40

[17] V. GULISANO, R. JIMENEZ-PERIS, M. PATIÑO-MARTÍNEZ, C. SORIENTE, AND P. VALDURIEZ. **StreamCloud: An Elastic and Scalable Data Streaming System**. *IEEE Transactions on Parallel and Distributed Systems*, **23**(12):2351–2365, 2012. 5, 11, 40

[18] GIUSEPPE DECANDIA, DENIZ HASTORUN, MADAN JAMPANI, GUNAVARDHAN KAKULAPATI, AVINASH LAKSHMAN, ALEX PILCHIN, SWAMINATHAN SIVASUBRAMANIAN, PETER VOSSHALL, AND WERNER VOGELS. **Dynamo: Amazon's Highly Available Key-value Store**. *SIGOPS Oper. Syst. Rev.*, **41**(6):205–220, October 2007. 9

[19] VOLDEMORT. **Voldemort**. 10

[20] APACHE. **Cassandra**. 10

[21] DYNOMITE. **Dynomite**. 10

[22] LARS GEORGE. *HBase: The Definitive Guide*. O'Reilly Media, 2011. 10

[23] HYPERTABLE. **Hypertable**. 10

[24] APACHE. **CouchDB**. 10

[25] MONGODB. **MongoDB**. 10

[26] APACHE. **Storm**. 10

[27] APACHE. **Samza**. 10

[28] WSO2. **WSO2**. 10

[29] TOM WHITE. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009. 35

[30] KONSTANTIN SHVACHKO, HAIRONG KUANG, SANJAY RADIA, AND ROBERT CHANSLER. **The Hadoop Distributed File System**. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, page 1–10, USA, 2010. IEEE Computer Society. 35

[31] JEFFREY DEAN AND SANJAY GHEMAWAT. **MapReduce: Simplified Data Processing on Large Clusters**. *Commun. ACM*, **51**(1):107–113, January 2008. 36

[32] VINOD KUMAR VAVILAPALLI, ARUN C. MURTHY, CHRIS DOUGLAS, SHARAD AGARWAL, MAHADEV KONAR, ROBERT EVANS, THOMAS GRAVES, JASON LOWE, HITESH SHAH, SIDDHARTH SETH, AND ET AL. **Apache Hadoop YARN: Yet Another Resource Negotiator**. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery. 36

[33] PATRICK HUNT, MAHADEV KONAR, FLAVIO P. JUNQUEIRA, AND BENJAMIN REED. **ZooKeeper: Wait-free Coordination for Internet-scale Systems**. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. 36

[34] PROMETHEUS. **Prometheus**. 49

[35] NAKUL MANCHANDA AND KARAN ANAND. **Non-Uniform Memory Access (NUMA)**. 50, 51

[36] CHRISTOPH LAMETER. **NUMA (Non-Uniform Memory Access): An Overview**. *Queue*, **11**(7):40:40–40:51, July 2013. 50

[37] SAS. **ETL - What it is and why it matters**. 57

[38] HORTONWORKS. **Determine YARN and MapReduce Memory Configuration Settings**. 64, 83

[39] J.E. BARLETT, J. KOTRLIK, AND C. HIGGINS. **Organizational Research: Determining Appropriate Sample Size in Survey Research**. *Information Technology, Learning, and Performance Journal*, **19**, 01 2001. 74

[40] MEDIAWIKI. **Ext4 Howto**. 86

[41] SCOTT T. LEUTENEGGER AND DANIEL DIAS. **A Modeling Study of the TPC-C Benchmark**. *SIGMOD Rec.*, **22**(2):22–31, June 1993. 96, 110

7. REFERENCIAS

[42] AINHOA AZQUETA-ALZÚAZ, MARTA PATIÑO MARTINEZ, IVAN BRONDINO, AND RICARDO JIMENEZ-PERIS. **Massive Data Load on Distributed Database Systems over HBase**. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '17, pages 776–779, Piscataway, NJ, USA, 2017. IEEE Press. 120

[43] RICARDO JIMÉNEZ-PERIS, FRANCISCO J. BALLESTEROS, AINHOA AZQUETA-ALZÚAZ, PAVLOS KRANAS, DIEGO BURGOS, AND PATRICIO MARTÍNEZ. **Parallel Efficient Data Loading**. In SLIMANE HAMMOUDI, CHRISTOPH QUIX, AND JORGE BERNARDINO, editors, *Proceedings of the 8th International Conference on Data Science, Technology and Applications, DATA 2019, Prague, Czech Republic, July 26-28, 2019*, pages 465–469. SciTePress, 2019. 120

[44] AINHOA AZQUETA-ALZÚAZ, IVAN BRONDINO, MARTA PATIÑO-MARTÍNEZ, AND RICARDO JIMÉNEZ-PERIS. **Load balancing for Key Value Data Stores**. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 506–509, 2017. 137

[45] APACHE FLINK. **Checkpointing**. 140

[46] KAFKA. **Kafka**. 140

[47] RABBITMQ. **Rabbitmq**. 140

[48] LESLIE LAMPORT. **Using Time Instead of Timeout for Fault-Tolerant Distributed Systems.** *ACM Trans. Program. Lang. Syst.*, **6**(2):254–280, April 1984. 144

[49] FRED B. SCHNEIDER. **Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial**. *ACM Comput. Surv.*, **22**(4):299–319, December 1990. 144

[50] MAGDALENA BALAZINSKA, HARI BALAKRISHNAN, SAMUEL R. MADDEN, AND MICHAEL STONEBRAKER. **Fault-Tolerance in the Borealis Distributed Stream Processing System**. *ACM Trans. Database Syst.*, **33**(1), March 2008.

[51] INTEL. **HiBench Benchmark**. 155

[52] MARTA PATIÑO-MARTÍNEZ AND AINHOA AZQUETA-ALZÚAZ. **Data Streaming for Appliances**. In VÍCTOR MÉNDEZ MUÑOZ, DONALD FERGUSON, MARKUS HELFERT, AND CLAUS PAHL, editors, *Proceedings of the 9th International Conference on Cloud Computing and Services Science, CLOSER 2019, Heraklion, Crete, Greece, May 2-4, 2019*, pages 672–678. SciTePress, 2019. 188

[53] MARTA PATIÑO-MARTÍNEZ AND AINHOA AZQUETA-ALZÚAZ. **Dynamic Data Streaming for an Appliance**. In SLIMANE HAMMOUDI, CHRISTOPH QUIX, AND JORGE BERNARDINO, editors, *Proceedings of the 8th International Conference on Data Science, Technology and Applications, DATA 2019, Prague, Czech Republic, July 26-28, 2019*, pages 470–477. SciTePress, 2019. 188

[54] VALERIO VIANELLO, MARTA PATIÑO-MARTÍNEZ, AINHOA AZQUETA-ALZÚAZ, AND RICARDO JIMENEZ-PÉRIS. **Cost of Fault-Tolerance on Data Stream Processing**. In GABRIELE MENCAGLI, DORA B. HERAS, VALERIA CARDELLINI, EMILIANO CASALICCHIO, EMMANUEL JEANNOT, FELIX WOLF, ANTONIO SALIS, CLAUDIO SCHIFANELLA, RAVI REDDY MANUMACHU, LAURA RICCI, MARCO BECCUTI, LAURA ANTONELLI, JOSÉ DANIEL GARCIA SANCHEZ, AND STEPHEN L. SCOTT, editors, *Euro-Par 2018: Parallel Processing Workshops*, pages 17–27, Cham, 2019. Springer International Publishing. 188

7. REFERENCIAS

Apéndice A

Inyector de carga distintas configuraciones

Tabla A.1: Tamaño de bloque y número de iteraciones de la tabla Customer

		AMD						XEON					
	Tamaño	Predefinido		yarn-util.py		Inyector		Predefinido		yarn-util.py		Inyector	
WHs	(GB)	Bloque	#Itera.	Bloque	#Itera.	Bloque	#Itera.	Bloque	#Itera.	Bloque	#Itera.	Bloque	#Itera.
1000	13	128	1	342	1	112	1	128	1	256	1	135	1
3000	39	128	4	1024	1	336	1	128	4	677	1	404	1
6000	77	128	8	2021	1	663	1	128	8	1337	1	797	1

Tabla A.2: Tamaño de bloque y número de iteraciones de la tabla Stock

		AMD						XEON					
	Tamaño	Predefinido		yarn-util.py		Inyector		Predefinido		yarn-util.py		Inyector	
WHs	(GB)	Bloque	#Itera.	Bloque	#Itera.	Bloque	#Itera.	Bloque	#Itera.	Bloque	#Itera.	Bloque	#Itera.
1000	28	128	3	736	1	241	1	128	3	512	1	290	1
3000	84	128	9	2206	1	723	1	128	9	1458	1	869	1
6000	169	128	17	1480	3	1455	1	128	17	1467	2	1749	1

Tabla A.3: Tamaño de bloque y número de iteraciones de la tabla Order_Line

		AMD						XEON					
	Tamaño	Predefinido		yarn-util.py		Inyector		Predefinido		yarn-util.py		Inyector	
WHs	(GB)	Bloque	#Itera.	Bloque	#Itera.	Bloque	#Itera.	Bloque	#Itera.	Bloque	#Itera.	Bloque	#Itera.
1000	19	128	2	551	1	164	1	128	2	512	2	218	1
3000	62	128	6	1628	1	534	1	128	6	1077	2	642	1
6000	125	128	13	1642	2	1075	1	128	13	1085	4	1211	1

A. INYECTOR DE CARGA DISTINTAS CONFIGURACIONES

Apéndice B

Inyector de carga distintos clusters

Tabla B.1: Tamaño de bloque y número de iteraciones para las tablas Customer, Stock y Order_Line en el clúster AMD

	Customer			Stock			Order_Line		
	Tamaño	Blo.	Ite.	Tamaño	Blo.	Ite.	Tamaño	Blo.	Ite.
1000	13	112	1	28	241	1	19	164	1
3000	39	336	1	84	723	1	62	534	1
6000	77	663	1	169	1455	1	125	1075	1
12000	154	1326	1	337	1450	2	251	1080	2
18000	231	1988	1	506	1452	3	379	1631	2

Tabla B.2: Tamaño de bloque y número de iteraciones para las tablas Customer, Stock y Order_Line en el clúster XEON

	Customer			Stock			Order_Line		
	Tamaño	Blo.	Ite.	Tamaño	Blo.	Ite.	Tamaño	Blo.	Ite.
1000	13	135	1	28	290	1	19	218	1
3000	39	404	1	84	869	1	62	642	1
6000	77	797	1	169	1749	1	125	1211	1
18000	231	1195	2	506	1745	3	379	1960	2

Tabla B.3: Tamaño de bloque y número de iteraciones para las tablas Customer, Stock y Order_Line en el clúster Bullion

	Customer			Stock			Order_Line		
	Tamaño	Blo.	Ite.	Tamaño	Blo.	Ite.	Tamaño	Blo.	Ite.
1000	13	135	1	28	404	1	19	303	1
3000	39	404	1	84	1212	1	62	895	1
6000	77	797	1	169	1218	2	125	1803	1
18000	231	1666	2	506	1825	4	379	1823	3
36000	461	1666	4	1022	1825	8	764	1823	6

B. INYECTOR DE CARGA DISTINTOS CLUSTERS

Apéndice C

Equilibrado de datos uniforme

Tabla C.1: Distribución de los datos obtenidos por el histograma antes de ejecutar el Equilibrado de Datos Uniforme, regiones 1 a 10.

Tabla	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Warehouse	168	31	49	79	137	175	105	202	34	31
District	119	753	3267	938	379	316	2427	2180	374	146
Item	3074	2245	578	6833	4365	1033	2	3524	468	4955
New_Order	1218068	217446	778193	108815	2247208	55427	257258	2837997	113887	1908806
Order	2272016	3115204	651130	701104	7385853	810315	610417	2914886	3099448	33865
History	2461439	2458723	2107158	2110857	2109308	2462099	2108432	2110122	2460649	2111803
Customer	902925	2134841	857825	6723965	3202420	2724608	672520	601536	554403	960354
Stock	12221771	10151075	7677441	8137640	9336215	2081781	22542845	8502821	18499271	10504473
Order_Line	2516996	15932545	16356251	5183817	13835232	2219333	1122592	11438630	6839513	3030719

Tabla C.2: Distribución de los datos obtenido por el histograma antes de ejecutar el Equilibrado de Datos Uniforme, regiones 11 a 20

Tabla	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20
Warehouse	24	42	84	234	30	9	15	153	66	13
District	276	1428	1111	1126	628	390	2344	567	86	58
Item	667	2661	2104	2103	7762	2161	1712	860	1721	1032
New_Order	388791	1369772	920702	963521	569259	511139	43241	675182	447745	1136403
Order	2277022	422587	492116	9721682	691686	2158964	183209	980775	1677382	881684
History	2109810	2461813	2108366	2110615	2460195	2463128	2458311	2462824	2461055	2105753
Customer	2094835	7188690	769933	611675	405208	7754305	701363	978361	5004523	24592
Stock	1615419	1857208	4157102	2507572	13219831	1782811	195686	31182657	1628407	5282284
Order_Line	5443015	14813779	25416769	19035098	13308143	8754915	3544833	93788180	15771092	2759762

Tabla C.3: Distribución de los datos obtenido por el histograma antes de ejecutar el Equilibrado de Datos Uniforme, regiones 21 a 30

Tabla	R21	R22	R23	R24	R25	R26	R27	R28	R29	R30
Warehouse	159	43	8	25	165	110	49	176	29	32
District	455	236	48	711	147	237	354	176	1132	283
Item	1125	3141	1002	2430	883	1579	5445	1462	2250	4511
New_Order	321972	250892	17749	453942	146125	130266	1363970	367450	484416	363797
Order	3061042	4585768	2388351	1632787	6159590	2399681	1298623	375834	1040502	2172340
History	2109587	2462775	2111041	2107722	2108868	2107221	2112567	2110031	2460533	2463901
Customer	5716792	2582292	556848	6924346	144560	631329	2988731	5645855	3054999	378899
Stock	7353794	835215	2418248	19221854	2338319	14397228	7892301	5465069	10650595	468444
Order_Line	49264393	35860241	49462951	13706736	29701662	7812600	6641787	16969767	14865843	10955059

Tabla C.4: Distribución de los datos obtenido por el histograma antes de ejecutar el Equilibrado de Datos Uniforme, regiones 31 a 40.

Tabla	R31	R32	R33	R34	R35	R36	R37	R38	R39	R40
Warehouse	6	22	14	65	19	73	55	169	34	66
District	109	116	197	2560	1697	418	324	921	787	179
Item	993	269	627	850	1320	5886	2251	466	4262	9388
New_Order	162930	166470	934408	215749	798064	436665	874596	1454475	646639	640565
Order	2964180	1820512	778699	278393	897498	828171	4008385	467152	2257699	9503448
History	2462026	2108350	2109732	2459995	2460567	2109763	2107107	2109061	2109418	2107275
Customer	1943110	607727	1176266	141895	168187	3814807	2430682	3385752	2554980	283061
Stock	2487918	2633599	1150323	27407771	6090076	1874398	462406	18376	9692966	4056790
Order_Line	10675397	59179560	11073483	8377976	21490388	960249	11181	9812310	121495735	5131187

Tabla C.5: Distribución de los datos obtenido por el histograma después de ejecutar el Equilibrado de Datos Uniforme, regiones 1 a 10.

Tabla	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Warehouse	75	75	75	75	75	75	75	75	75	75
District	750	750	750	750	750	750	750	750	750	750
Item	2500	2500	2500	2500	2500	2500	2500	2500	2500	2500
New_Order	683887	680702	681866	680000	680000	678423	676403	679259	677651	676665
Order	2259999	2259993	2259230	2251152	2256293	2255873	2010420	2254246	2259366	2251925
History	2110031	2462026	2109810	2458311	2459995	2107275	2108868	2109732	2110857	2460649
Customer	2259433	2260000	2260000	2260000	2259938	2260000	2255685	1962293	2254346	2250781
Stock	7502376	7510000	7509634	7510000	7502966	7503355	7510000	7510000	7510000	7504473
Order_Line	19116769	19115098	19121092	19121662	19120000	19123779	19120000	19120000	19120000	19123026

Tabla C.6: Distribución de los datos obtenido por el histograma antes de ejecutar el Equilibrado de Datos Uniforme, regiones 11 a 20

Tabla	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20
Warehouse	75	75	75	75	75	75	75	75	75	75
District	750	750	750	750	750	750	750	750	750	750
Item	2500	2500	2500	2500	2500	2500	2500	2500	2500	2500
New_Order	679999	683970	680000	680000	678068	684596	675722	680000	680763	676639
Order	2259180	2252225	2259652	2256784	2251073	2251143	2250413	2254045	2257860	2255077
History	2111803	2461055	2463128	2460567	2463901	2109061	2105753	2460533	2461813	2458723
Customer	2253110	2260000	2260000	2256792	2260000	2260000	2260000	2254523	254807	2260000
Stock	7505979	7502283	7508977	7509831	7510000	510000	7508531	7510000	7506215	7510000
Order_Line	19120000	19122369	19122951	19120000	19114597	19120000	19121787	19120000	19116251	19115232

Tabla C.7: Distribución de los datos obtenido por el histograma antes de ejecutar el Equilibrado de Datos Uniforme, regiones 21 a 30

Tabla	R21	R22	R23	R24	R25	R26	R27	R28	R29	R30
Warehouse	75	75	75	75	75	75	75	75	75	75
District	750	750	750	750	750	750	750	750	750	750
Item	2500	2500	2500	2500	2500	2500	2500	2500	2500	2500
New_Order	680000	680000	677208	676252	684408	680565	680000	679646	488194	680927
Order	2255362	2260000	2260000	2257889	2260000	2251828	2257723	2260000	2256157	2255483
History	2107107	2110615	2107722	2107158	2109763	2462099	2110122	2109418	2108432	2461439
Customer	2260000	2257760	2260000	2256144	2250247	2255334	2260000	2250922	2250682	2256883
Stock	7510000	7507640	7503922	7510000	7501467	7509271	7502301	7509213	7510000	7214589
Order_Line	19120000	19115366	19123386	19120000	19120000	19120479	19119999	19120388	19114682	19115574

Tabla C.8: Distribución de los datos obtenido por el histograma después de ejecutar el Equilibrado de Datos Uniforme, regiones 31 a 40.

Tabla	R31	R32	R33	R34	R35	R36	R37	R38	R39	R40
Warehouse	75	75	75	75	75	75	75	75	75	75
District	750	750	750	750	750	750	750	750	750	750
Item	2500	2500	2500	2500	2500	2500	2500	2500	2500	2500
New_Order	680000	677609	680000	680000	680000	680000	682006	677030	681542	680000
Order	2257787	2250602	2257589	2260000	2259460	2258251	2252836	2257839	2252202	2253043
History	2112567	2460195	2107221	2108350	2462824	2109587	2108366	2111041	2462775	2109308
Customer	2260000	2256493	2255855	2260000	2258731	2260000	2254305	2260000	2260000	2254936
Stock	7507441	7503892	7510000	7510000	7510000	7501771	7502821	7510000	7502657	7508395
Order_Line	19120000	19120000	19117707	19115843	19120241	19120000	18898844	19120000	19122597	19120000