

# 4

## PROCESAMIENTO DE DATOS POR LOTES

Como ya hemos ido avanzando en capítulos anteriores, cuando nos planteamos como podemos tratar los datos en su transformación y conversión en información, tenemos la disyuntiva entre el procesamiento por lotes y en tiempo real. La Tabla 4-1 compara las características de ambos.

Procesamiento por lotes	Procesamiento en tiempo real
<ul style="list-style-type: none"><li>• Los datos deben ser contextualizados en un marco temporal para ser tratados de forma conjunta</li><li>• Durante el procesamiento, los datos pueden ser almacenados temporalmente en varias ocasiones</li><li>• Los procesos de tratamiento suelen estar planificados o pueden desencadenarse por algún evento</li><li>• Las operaciones y análisis de negocio que dependen de los datos deben adaptarse a la ventana de ejecución de los procesos o viceversa</li><li>• Las transformaciones sobre los datos pueden ser exhaustivas</li></ul>	<ul style="list-style-type: none"><li>• Es necesario gestionar eventos de forma individual, tan pronto como se generan</li><li>• Los datos asociados a los eventos son gestionados en memoria, con una persistencia mínima o nula</li><li>• El procesamiento de eventos se realiza de forma continua e ininterrumpida, con la mínima demora</li><li>• La captura, transformación y análisis del dato conforman un flujo continuo desde los orígenes de los eventos hasta las aplicaciones consumidoras finales</li><li>• Un tratamiento intensivo del dato impactará en el tiempo de entrega</li></ul>

Tabla 4-1. Características del procesamiento por lotes y en tiempo real

Si bien existen cada vez más tendencias y tecnologías para unificar ambas aproximaciones, vamos a tratar cada una por separado para entender mejor sus requerimientos y necesidades. En este capítulo abordaremos básicamente la gestión por lotes, mientras que en el siguiente estudiaremos el procesamiento en tiempo real y como combinar ambos de una forma efectiva.

## 4.1 EXTRACCIÓN, TRANSFORMACIÓN Y CARGA

De forma genérica, denominamos **extracción, transformación y carga (ETL, Extract, Transform, Load)** al conjunto de etapas de procesamiento que se encargan de obtener datos de los sistemas origen de la organización con el fin de combinarlos y tratarlos de cara a su persistencia, uso y explotación posterior, normalmente de carácter analítico. La Figura 4-1 esquematiza esta idea; entre los orígenes y los destinos de los datos disponemos de un **área intermedia**, con su propia tecnología, recursos de cómputo y almacenamiento, donde se llevan a cabo estas etapas.

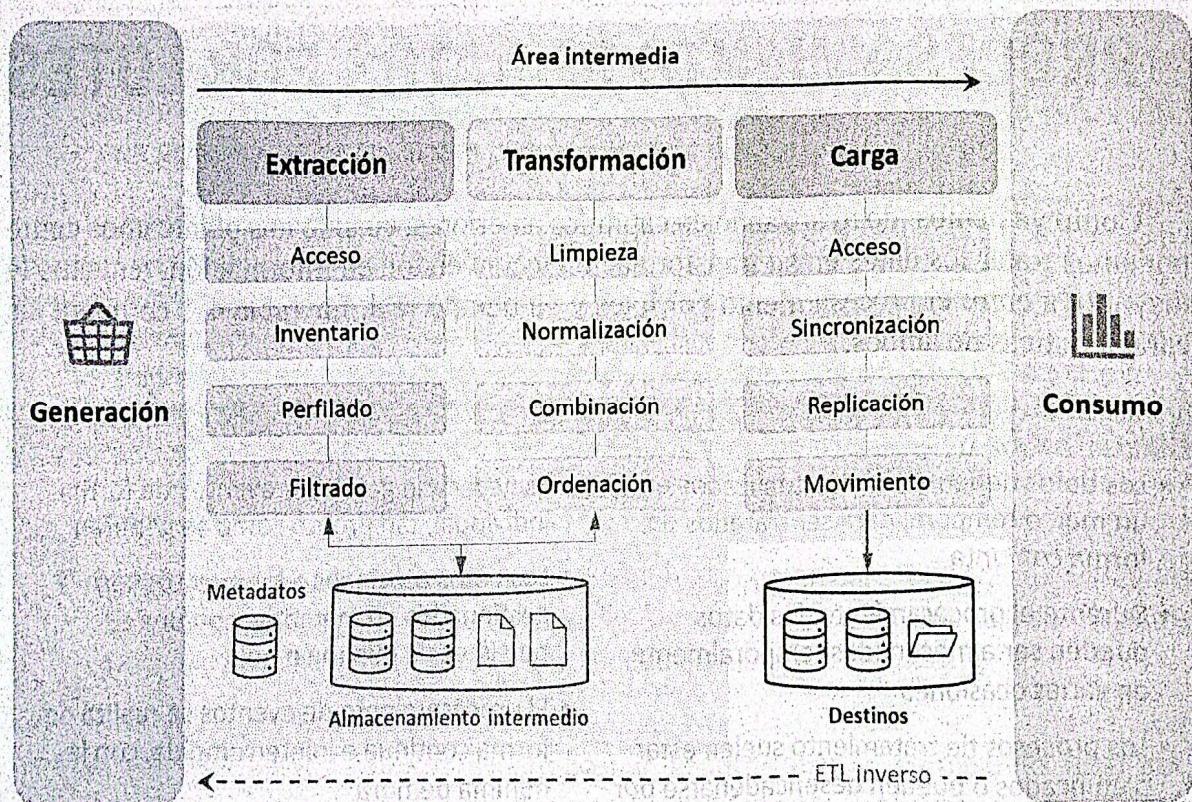


Figura 4-1. Procesos de extracción, transformación y carga.

Desde un punto de vista técnico y organizativo, estas etapas se podrían considerar como una parte de la **ingeniería de datos**, entendida esta como la gestión integral de la infraestructura de datos, con una visión transversal que cubre la modelización, la transformación, el gobierno y el aseguramiento de la calidad del dato, lindando ya con el consumo y el análisis de la información.

### 4.1.1 Extracción

La extracción tiene como objetivo la recopilación inicial de todos los datos de interés<sup>90</sup>, susceptibles de ser posteriormente analizados, con el mínimo impacto en los sistemas origen que los contienen. No olvidemos que la mayoría de estos orígenes son sistemas operacionales en producción, por lo que la extracción no debería interferir en sus tareas. Como norma general se debería acceder a cada sistema, extraer los datos y almacenarlos directamente en la **zona de almacenamiento** del área intermedia. Esta persistencia temporal permite recuperar los datos extraídos en el caso en que alguna etapa posterior falle, pudiendo reiniciar todo el proceso sin necesidad de volver a incidir en los orígenes.

En términos de frecuencia de acceso, lo habitual es hacerlo por intervalos de tiempo, realizando extracciones diarias, semanales, mensuales, etc. Otra posibilidad es ligar la extracción a la acumulación de un número determinado de registros, a modo de gestión de eventos por lotes.

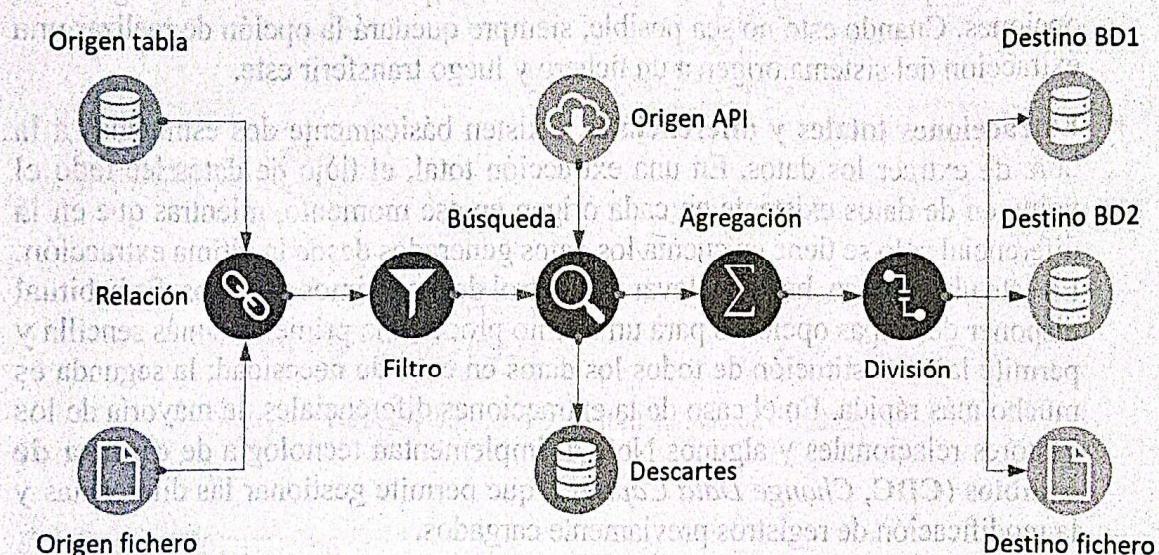


Figura 4-2. Ejemplo de flujo ETL. Cada círculo es un proceso, con sus entradas y, si aplican, salidas.

90 Aunque lo estamos obviando, es fundamental, tras un análisis de requerimientos, un inventario que establezca y catalogue los distintos orígenes de datos.

La etapa de extracción es el comienzo de lo que se denomina un **flujo ETL<sup>91</sup>** (*ETL pipeline*); es decir, una secuencia de procesos por el que van pasando los datos, siendo la salida de un proceso la entrada del siguiente. Estos procesos son los encargados de ir moviendo los datos desde los orígenes hasta sus destinos, con toda una serie de manipulaciones y transformaciones intermedias. La Figura 4-2 muestra un ejemplo. El número de estos flujos que puede tener una organización puede llegar al orden de cientos o miles, con complejidades variables y distintos gastos de puesta en producción y planificación.

Hay una serie de consideraciones y tareas a la hora de concebir e implementar esta etapa:

- **Variedad de orígenes.** Las fuentes de datos pueden ser heterogéneas, tanto en ubicación (locales, en la nube), formato (estructurado, semiestructurado, no estructurado, comprimido), tecnología (bases de datos propietarias, ERP, CRM, archivos de código abierto, API) y propiedad (internas, externas). La tecnología de ETL empleada debe proporcionar conectores que cubran una amplia gama de opciones. Cuando esto no sea posible, siempre quedará la opción de realizar una extracción del sistema origen a un fichero y luego transferir este.
- **Extracciones totales y diferenciales.** Existen básicamente dos estrategias a la hora de extraer los datos. En una extracción total, el flujo de datos lee todo el volumen de datos existente en cada origen en ese momento, mientras que en la diferencial sólo se tiene en cuenta los datos generados desde la última extracción. En este último caso, hay que llevar un control de los últimos registros. Es habitual disponer de ambas opciones para un mismo proceso: la primera es más sencilla y permite la reconstitución de todos los datos en caso de necesidad; la segunda es mucho más rápida. En el caso de las extracciones diferenciales, la mayoría de los gestores relacionales y algunos NoSQL implementan tecnología de **captura de cambios (CDC, Change Data Capture)**, que permite gestionar las diferencias y la modificación de registros previamente cargados.
- **Inventariado y perfilado.** Además de inventariar y documentar los orígenes, es fundamental realizar un **perfil de los datos (data profiling)**. Este consiste en un análisis y auditoría de la estructura y la calidad de los datos, permitiendo obtener estadísticas, identificar valores anómalos, inconsistencias en la codificación o posibles relaciones entre estructuras de orígenes distintos. Esto es fundamental de cara a implementar acciones correctoras en la etapa de transformación.

91 Estos flujos ETL se pueden considerar una subcategoría de lo que genéricamente se conoce como flujos de datos (*data pipelines*). Mientras los primeros tienen un alcance y un contexto muy definido, sus mayores hacen referencia a cualquier manipulación alrededor del dato, con independencia de su origen y destino, pudiendo involucrar también procesos de modelización predictiva, despliegue en producción o monitorización. Hablaremos de estos flujos de datos más adelante en el libro.

Una vez que hemos extraído los datos al área intermedia, los hemos analizado y efectuado un posible filtrado inicial, es el momento de implementar procesos de transformación en el flujo ETL.

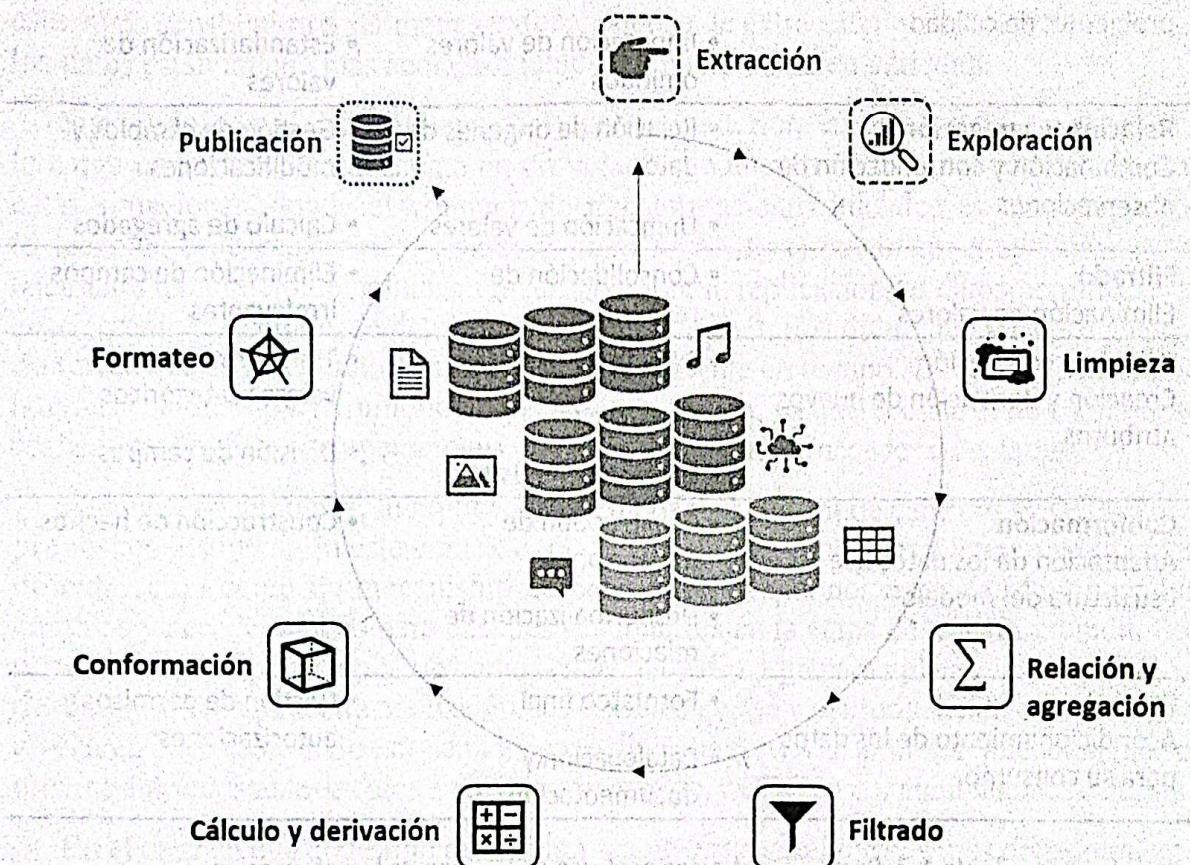


Figura 4-3. Tareas en la transformación de datos.

## 4.1.2 Transformación

La amplitud de la etapa de transformación es variable, y depende de varios factores. Mientras que en un *data lake* sería inicialmente mínima o inexistente, en un *data warehouse* estaría poblada de actividades. Esto es consistente con la idea de almacenar los datos en crudo en el primero, y de hacerlo de forma conformada y estandarizada en el segundo.

Sin embargo, no perdamos de vista que en el caso del *data lake*, las necesidades de transformación empiezan precisamente una vez que los datos han aterrizado en el repositorio. Por este motivo, sería más propio hablar de **extracción, carga y transformación (ETL, Extraction, Loading, Transformation)** en este caso.

Sea donde sea, el objetivo es acondicionar los datos para su consumo, medie o no un punto de almacenamiento. La Figura 4-3 muestra las principales tareas que suelen integrar esta etapa, mientras que la Tabla 4-2 las detalla de forma orientativa a nivel de operaciones.

Tareas	Operaciones
<b>Limpieza</b> Detección y corrección de problemas de calidad	<ul style="list-style-type: none"> <li>• Gestión de valores erróneos</li> <li>• Imputación de valores omitidos</li> <li>• Eliminación de duplicados</li> <li>• Estandarización de valores</li> </ul>
<b>Relación y agregación</b> Combinación y consolidación de observaciones	<ul style="list-style-type: none"> <li>• Relación de orígenes de datos</li> <li>• Unificación de valores</li> <li>• Gestión de cambios y modificaciones</li> <li>• Cálculo de agregados</li> </ul>
<b>Filtrado</b> Eliminación de valores	<ul style="list-style-type: none"> <li>• Consolidación de registros</li> <li>• Eliminación de campos irrelevantes</li> </ul>
<b>Cálculo y conversión</b> Creación y derivación de nuevos atributos	<ul style="list-style-type: none"> <li>• Cálculo de métricas</li> <li>• Codificación</li> <li>• División de registros</li> <li>• Transformación a valores categóricos</li> <li>• División de campos</li> </ul>
<b>Conformación</b> Adaptación de los datos a la estructura del modelo <sup>92</sup>	<ul style="list-style-type: none"> <li>• Construcción de dimensiones</li> <li>• Desnormalización de relaciones</li> <li>• Construcción de hechos</li> </ul>
<b>Formateo</b> Acondicionamiento de los datos para su consumo	<ul style="list-style-type: none"> <li>• Formateo final</li> <li>• Catalogación y documentación</li> <li>• Gestión de permisos y autorizaciones</li> </ul>

Tabla 4-2. Operaciones típicas por tarea en la etapa de transformación.

De forma similar a la etapa de extracción, cada una de estas tareas tiene su propia persistencia, incluso a nivel de operación. Esto permite plantear las transformaciones como pasos aditivos, facilitando la ejecución del flujo desde cualquier punto, sin necesidad de reiniciarlas por completo.

Hemos comentado que las operaciones de transformación, como las de extracción y carga, se llevan a cabo en un área intermedia por un sistema dedicado. Estas se pueden realizar empleando distintos lenguajes de programación e interrogación de propósito general (Python, Scala, SQL, etc.), o bien tecnologías más especializadas, que veremos más adelante. En este sentido, y siguiendo la tendencia de descomposición de aplicaciones en **microservicios**, cada vez es más frecuente el ensamblado de operaciones ejecutadas en sistemas dispersos y desacoplados, gestionados por distintos proveedores en la nube y sin reserva de recursos (*serverless*).

92 Esta tarea está muy ligada a arquitecturas de *data marts* dependientes, siguiendo el modelo de Kimball.

### 4.1.3 Carga

La etapa final en ETL es la de publicar los datos en su destino. Esto es, mover los datos del área intermedia al repositorio donde serán consumidos. Respecto a la conectividad, esta etapa acostumbra a ser menos exigente que la de extracción, ya que los destinos de los datos están mucho más acotados, tanto en número como en variedad.

Siguiendo con el razonamiento del apartado anterior, en el caso de un *data lake* la carga consistirá prácticamente en un volcado de nuevos datos en el repositorio. En estas situaciones, donde el volumen de transformaciones iniciales es muy pequeño, la tecnología de **replicación de datos** (*data replication*) puede ser muy interesante. Mediante un modelo de publicación-subscripción, la replicación de datos permite tener en sincronía estructuras de datos que residen en repositorios diferentes y heterogéneos, de forma que se mantienen consistentes prácticamente en tiempo real. Esto se consigue además con un impacto mínimo en los sistemas origen, ya que el control de cambios se realiza en el registro de operaciones (*log*), sin necesidad de acceder a las tablas<sup>93</sup>.

Aunque son similares, la replicación de datos y el control de cambios (CDC) tienen propósitos distintos. En este contexto, el CDC se utiliza fundamentalmente en la etapa de extracción como un mecanismo de detección, ya que nos permite identificar los cambios y acceder a ellos para después gestionarlos en la etapa de transformación. Por el contrario, la replicación la ubicamos en la etapa de carga para aplicar directamente en destino, y sin ninguna otra mediación, los cambios que se van produciendo en el origen o en el área intermedia<sup>94</sup>. En este sentido, es posible que convivan ambas; replicación para el movimiento constante de datos entre origen y destino, y CDC para ir auditando los cambios.

En el caso en que el destino de la carga sea un *data warehouse*, y más concretamente una serie de modelos multidimensionales, la etapa de carga será más densa en cuanto a operaciones. Como veímos en el capítulo sobre arquitecturas, estos modelos se basan en una serie de dimensiones conformadas, que requieren un mantenimiento ante la llegada de datos nuevos, especialmente en lo referente a la gestión de cambios.

Por ejemplo, si tenemos una dimensión Cliente con una colección de atributos sociodemográficos, ¿qué ocurre cuando un cliente se va a vivir a otra región en una fecha determinada? Nuestro sistema de extracción detectará el cambio que se habrá producido en un sistema operacional (CRM), la etapa de transformación habrá validado y normalizado el registro del cliente que contiene ese cambio y, por último, la etapa de carga tendrá que aplicarlo, trasladándolo a la tabla que contiene la dimensión. Este traslado, y por lo tanto la gestión del cambio, debería ser siempre una decisión de negocio: ¿queremos ver a ese cliente como si siempre hubiese vivido en la nueva región?, o bien, ¿nos interesa diferenciar el antes y el después? Sea cual sea la respuesta, esta tendrá un

93 Aunque típica de las bases de datos relacionales, la replicación de datos es también una característica de gestores NoSQL, como MongoDB o Apache Cassandra.

94 Por la necesidad de ir aplicando los cambios con una latencia mínima, las opciones de transformación de datos en la replicación suelen ser mínimas, de forma que las tablas de destino suelen ser espejos de la de origen.

impacto a la hora de consumir los datos, afectando incluso a consultas e informes ya existentes. La forma de gestionar estos cambios es un aspecto fundamental de la etapa de carga, y la veremos en el siguiente apartado cuando hablemos de modelos de datos.

De forma paralela a la extracción, las **cargas** en un *data warehouse* pueden ser **totales** (*full refresh*) o **incrementales**. Lo habitual es hacer una carga total inicial seguida de cargas periódicas incrementales, conteniendo solo los cambios. El uso posterior de una nueva carga total vendría condicionado por la necesidad de corregir errores importantes o de reiniciar el sistema por algún motivo.

Dos apuntes finales antes de concluir este apartado sobre ETL. El primero de ellos es sobre **ETL inverso** (*reverse ETL*). Cuando en el capítulo sobre arquitecturas representábamos los flujos de datos entre el área intermedia y los distintos repositorios (Figura 2-9, por ejemplo), estas uniones eran siempre bidireccionales. Desde el momento en que en el entorno de análisis tenemos la capacidad de generar nuevos datos, normalmente a través de modelización predictiva, puede existir la necesidad de transportar estos hacia atrás, siendo el destino ahora los sistemas operacionales. Por ejemplo, este puede ser el caso de un modelo de segmentación de clientes, que nos interesa tener en un ODS de atención al público y en el CRM operacional. El ETL inverso permite recorrer este camino, existiendo una serie de tecnologías al respecto<sup>95,96</sup>.

El segundo es sobre la **gestión de metadatos**. Todas las etapas y actividades que hemos contemplado son generadoras de una gran cantidad de metadatos de proceso, conteniendo información sobre la tipología y número de registros extraídos, transformados, rechazados, cargados, tiempos de ejecución de cada flujo, errores, etc. Estos metadatos se unen a los metadatos operacionales y técnicos, permitiendo dibujar todo el ciclo de vida de los datos y realizar análisis de impacto de los diferentes elementos. Toda área intermedia, con independencia de lo desacoplados que puedan estar sus servicios, debería contar con un **repositorio de metadatos** dirigido a los ingenieros de datos, con el fin de soportar tareas de monitorización, análisis comparativo y auditoría. Hablaremos más de metadatos y su gestión en el último capítulo del libro.

## 4.2 MODELADO DE DATOS Y GESTIÓN DE CAMBIOS

Es difícil hablar con cierta profundidad de procesos de transformación y carga sin tener en cuenta el modelo de datos de destino. En repositorios sin esquema (*schemaless*) esto carece de sentido, ya que será la aplicación consumidora la que dicte como es la

95 El nombre de ETL inverso se presta a confusión, ya que se asocia fácilmente con la idea de deshacer lo ya hecho. Por el contrario, la idea es la de establecer un flujo de datos bidireccional entre sistemas.

96 Como veremos en capítulos posteriores, otra opción para este escenario es la de desplegar el modelo de segmentación en producción e invocarlo desde las aplicaciones que lo necesiten. En este caso se publica el modelo, mientras que en el ETL inverso esto se hace con los datos que genera. Cada aproximación tiene sus propios casos de uso, así como ventajas e inconvenientes.

organización de los datos y los procesos de transformación que hay que llevar a cabo. Como ya sabemos, estos escenarios los encontramos principalmente en arquitecturas de *data lakes*, apoyados en sistemas de archivos distribuidos, bases de datos clave-valor y de documentos.

## 4.2.1 Modelos multidimensionales

Sin embargo, en un entorno de *data warehouse* necesitamos un modelo, más o menos fijo, que permita tipificar su contenido y acceso, garantizando que la información que contiene es veraz y se ajusta a un esquema que es público en la organización. Con cierta independencia de la arquitectura de referencia (Inmon versus Kimball), el **modelado multidimensional**, implementado a través de *data marts*, es la forma habitual de diseñar la capa de presentación de los datos a los usuarios.

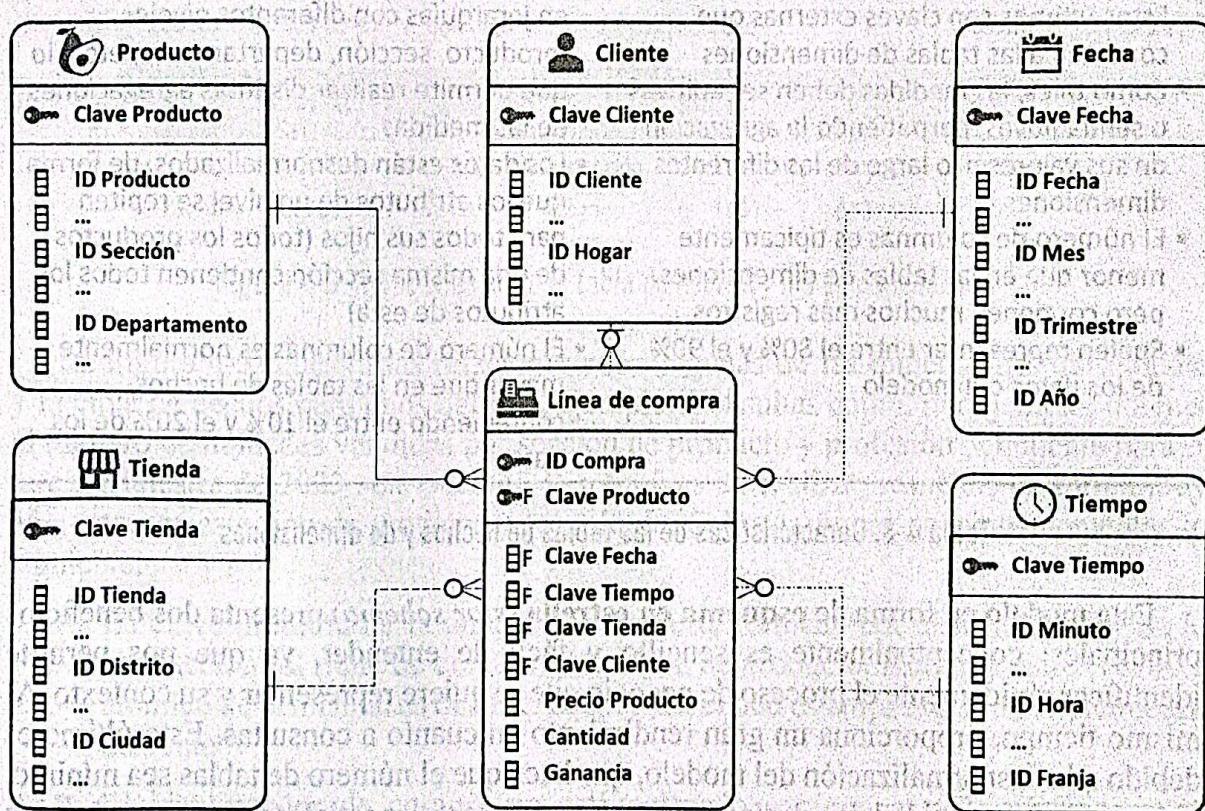


Figura 4-4. Modelo multidimensional en estrella.

La Figura 4-4 muestra un ejemplo de modelo multidimensional que representa el proceso de venta en un comercio minorista. En él se puede apreciar una **tabla de hechos central**, donde tenemos métricas sobre cada producto existente en una cesta de la compra, rodeada de cinco **tablas de dimensión**: fecha de la compra, hora, producto al que corresponde la línea de compra, tienda en la que tuvo lugar la adquisición y cliente que la realizó. El proceso de compra que queremos analizar podría contener más dimensiones, como empleado que realizó la venta o promoción asociada al producto.

## Tablas de hechos

- Un registro se corresponde con un evento que define el nivel de detalle con el que se mide el proceso (producto individual en una cesta de la compra)
- Cada registro se compone de un conjunto de campos numéricos con las medidas del evento (unidades vendidas, precio de venta, coste, descuento, etc.) y de una serie de campos que caracterizan al evento (fecha, tiempo, producto, tienda, etc.). Estos últimos son claves externas que conectan a las tablas de dimensiones
- Como tales, las medidas deben ser aditivas o semiaditivas, permitiendo la agregación de sus valores a lo largo de las diferentes dimensiones
- El número de columnas es típicamente menor que en las tablas de dimensiones, pero contienen muchos más registros
- Suelen representar entre el 80% y el 90% de los datos del modelo

## Tablas de dimensiones

- Proporcionan el contexto de cada evento en la tabla de hechos
- Cada tabla se compone de una clave primaria, que se enlaza a la tabla de hechos con el mismo nivel de detalle, y de un conjunto de atributos que describen las medidas (nombre del producto, formato, fabricante, sección a la que pertenece, etc.)
- Los atributos acostumbran a agruparse en jerarquías con diferentes niveles (producto, sección, departamento, etc.), lo que permite realizar distintas agregaciones de las medidas
- Los datos están desnormalizados, de forma que los atributos de un nivel se repiten para todos sus hijos (todos los productos de una misma sección contienen todos los atributos de esta)
- El número de columnas es normalmente mayor que en las tablas de hechos, conteniendo entre el 10% y el 20% de los datos

**Tabla 4-3.** Características de las tablas de hechos y de dimensiones

Este modelo en forma de **esquema en estrella** (*star schema*) presenta dos beneficios principales: conceptualmente es sencillo y fácil de entender, ya que nos permite identificar rápidamente el proceso de negocio que se quiere representar y su contexto. Al mismo tiempo, proporciona un gran rendimiento en cuanto a consultas. Esto último es debido a la desnormalización del modelo, que hace que el número de tablas sea mínimo, reduciendo la necesidad de relaciones (*joins*) entre estas para satisfacer una consulta, una operación siempre costosa<sup>97</sup>.

- 
- 97 Hay procesos de negocio en los que es necesario utilizar más de una tabla de hechos para representar las métricas de interés, por lo que esta no es siempre única.
- 98 Una alternativa al modelo en estrella es el llamado **modelo en copo de nieve** (*snowflake schema*). En este caso, las tablas de dimensiones no están desnormalizadas, separándose los atributos de baja cardinalidad a su propia tabla. Por ejemplo, la dimensión Producto se expandiría con una tabla para el nivel Sección, otra para el nivel Departamento, etc. Esto provoca un aumento del número de enlaces entre tablas que se necesitan para satisfacer una consulta, impactando en el rendimiento. Sin embargo, hay situaciones en las que es recomendable este tipo de modelado.

Hechos	ID Compra	Clave Producto	Clave Fecha	Clave Tiempo	Clave Tienda	Clave Cliente	Precio	Cantidad	Margen
	TR02672	4435623	230701	1423	654687	8876744	0,50€	5	1,73€
	TR02672	7776454	230701	1423	654687	8876744	22,3€	11	237,7€
	TR07453	6456575	230702	1401	653545	1111111	2,34€	2	3,01€
Producto	Clave Producto	ID Producto	Nombre Producto	Nombre Marca	ID Sección	Nombre Sección	ID Dato	Nombre Dato	
	4435623	SKU085558	Yogur Frutas A	Granja H	234	Yogures	2	Lácteos	
	7776454	SKU097568	Vino Blanco B	Bodegas C	476	Bodega	3	Bebidas	
Fecha	Clave Fecha	ID Fecha	Día Semana	ID Mes	Nombre Mes	ID Trimestre	Nombre Trimestre	ID Año	
	230701	2023-07-01	sábado	7	Julio	3	3 <sup>er</sup> Trm.	2023	
	230702	2023-07-02	domingo	7	Julio	3	3 <sup>er</sup> Trm.	2023	
Cliente	Clave Cliente	ID Cliente	Nombre Cliente	Profesión	Año Vinculación	ID Hogar	Código Postal		
	1111111	CC7899	Anónimo	Desconocida	Desconocido	1100110	00000		
	8876744		Xoan P.	Abogado	1997	6774434	15003		

Figura 4-5. Relaciones entre la tabla de hechos y las de dimensiones.

La Figura 4-5 muestra las relaciones entre algunas de las dimensiones del modelo y la tabla de hechos para unos datos de ejemplo. Consultas como, por ejemplo, obtener el número de unidades vendidas por sección de producto y profesión del cliente para el tercer trimestre de 2023 son, además de intuitivas de realizar, resueltas de forma muy eficiente por este tipo de modelos y tecnología. Sobre la figura podemos comentar lo siguiente:

- ▶ La clave primaria de la tabla de hechos está formada por el identificador de la transacción (el tique de compra) y la clave de producto. Esto nos marca el nivel atómico de la misma, es decir, el detalle con el que estamos registrando el proceso de venta<sup>99</sup>.
- ▶ El identificador de compra no está enlazado a ninguna tabla, tratándose de una **dimensión degenerada**. Su única función es registrar la integridad de la transacción y permitirnos realizar consultas que la involucren, como conteos y ratios.

99 El nivel atómico o de detalle de la tabla de hechos es una decisión de negocio, habida cuenta de que los datos existen en origen a ese nivel. Su elección es importante, ya que una vez establecido, su cambio exigiría la reelaboración del modelo y de los procesos ETL asociados. Si elegimos un nivel demasiado agregado, por ejemplo ventas mensuales de un producto por tienda, perderemos las dimensiones Tiempo y Cliente, viéndose también afectada la dimensión Fecha al perder sus atributos de mayor detalle. Es decir perdemos matiz en el análisis. Como regla general, siempre es recomendable registrar el mayor nivel de detalle disponible.

- ▀ La clave primaria de cada tabla de dimensión (y la correspondiente clave externa en la tabla de hechos) es un valor entero artificial, denominado **clave subrogada**, sin ningún sentido de negocio, generado y mantenido por los procesos ETL que transforman y cargan los datos. Hay varios motivos que justifican su existencia. El primero es poder lidiar con situaciones en las que no existe una clave natural del registro. Este es el caso de las transacciones anónimas, en las que no conocemos al cliente, ya sea por qué no la presenta o no dispone de tarjeta de fidelización. En este caso no existe un identificador, pero necesitamos poder relacionar estas compras de alguna manera con la tabla de hechos. Una situación parecida la podemos encontrar, quizás no en este tipo de modelo, cuando hay fechas que todavía no se han concretado, por ejemplo en la recepción o entrega de pedidos («por entregar», «pendiente de recibir», etc.). Otro motivo, aunque menos importante hoy en día, es el de hacer que las relaciones entre tablas se realicen mediante números enteros, ya que ocupan menos y son más eficientes en la indexación y los enlaces. Un tercer motivo es la gestión de los cambios, y lo veremos en el siguiente apartado.
- ▀ Las tablas de dimensiones contienen **niveles jerárquicos** que se van agregando de izquierda a derecha. Cada nivel consta de un identificador de nivel (ID fecha) y un número variable de columnas representando atributos (día de la semana, día del mes, indicador de festivo, indicador de fin de semana, etc.). De esta forma es muy sencillo y eficiente navegar por estas jerarquías a la hora de hacer consultas, pudiendo agregar y desagregar las métricas entre niveles.
- ▀ Los datos en las dimensiones están desnormalizados. Por ejemplo, las fechas 2023-07-01 y 2023-07-02 comparten los mismos valores para las columnas a partir del identificador de mes.
- ▀ En la tabla de hechos hay tres medidas. Mientras cantidad y margen son **medidas aditivas** para todas las dimensiones, ya que tiene sentido sumarlas a lo largo de cualquier eje, el precio unitario es una **medida no aditiva**, ya que su suma no produce un valor con sentido en ninguna dimensión. Por el contrario, si tiene sentido otro tipo de agregación, como el valor mínimo, el máximo o el promedio. Medidas no aditivas acostumbran a ser los porcentajes, ratios y valores promedios. También nos podemos encontrar **medidas semiaditivas**, que son aquellas que se pueden sumar a través de unas dimensiones pero no de otras. Por ejemplo, el número de unidades de un producto en un almacén es algo que podemos sumar a lo largo de una dimensión Almacén o Producto, pero no a través de una dimensión temporal.

Un aspecto importante en el mantenimiento de este tipo de modelos es como gestionar los cambios. Estos pueden ser **estructurales**, consecuencia de la adición de nuevas dimensiones, nuevos atributos a las dimensiones existentes o nuevas medidas a la tabla de hechos, debidos a **requerimientos de negocio**, como un cambio en el nivel de detalle que debe reflejar en el modelo, o **cambios en los datos**, consecuencia de la evolución natural del negocio y su entorno. Vamos a ver cómo podemos manejar estos últimos y el papel que el sistema de ETL tiene en ello.

## 4.2.2 Cambios en los datos y gestión de la historia

Cuando en el Apartado 4.1.3 hablábamos sobre carga de datos en el *data warehouse*, mencionábamos la necesidad de tener que gestionar cambios en los atributos de las dimensiones: un cliente cambia de lugar de residencia, un producto es reasignado a un nuevo departamento, o una tienda es renombrada como parte de una nueva estrategia de marca. Toda dimensión, exceptuando las temporales, que son prácticamente estáticas, son susceptibles de cambiar.

El principal problema con esto es la frecuencia con la que se pueden producir estos cambios. Si los atributos de una dimensión cambian de forma esporádica, no recurrente, entonces decimos que es una **dimensión que varía lentamente en el tiempo (SCD)**, *Slowly Changing Dimension*, y las distintas estrategias de gestión se numeran a modo de tipos (SCD-1, SCD-2, etc.). Es importante resaltar que estas estrategias son a nivel de atributo, de forma que en una misma dimensión podemos tener atributos gestionados de forma diferente.

Tipo	Ejemplo																																			
<b>SCD-1</b>																																				
Actualización del registro	<table border="1"> <thead> <tr> <th>Clave Cliente</th><th>ID Cliente</th><th>Nombre Cliente</th><th>Profesión</th><th>Año Vinculación</th><th>ID Hogar</th><th>Código Postal</th></tr> </thead> <tbody> <tr> <td>8876744</td><td>CC7899</td><td>Juan P.</td><td>Abogado</td><td>1997</td><td>6774434</td><td>15003</td></tr> </tbody> </table> <table border="1"> <thead> <tr> <th>Clave Cliente</th><th>ID Cliente</th><th>Nombre Cliente</th><th>Profesión</th><th>Año Vinculación</th><th>ID Hogar</th><th>Código Postal</th></tr> </thead> <tbody> <tr> <td>8876744</td><td>CC7899</td><td>Xoan P.</td><td>Abogado</td><td>1997</td><td>6774434</td><td>15003</td></tr> </tbody> </table>	Clave Cliente	ID Cliente	Nombre Cliente	Profesión	Año Vinculación	ID Hogar	Código Postal	8876744	CC7899	Juan P.	Abogado	1997	6774434	15003	Clave Cliente	ID Cliente	Nombre Cliente	Profesión	Año Vinculación	ID Hogar	Código Postal	8876744	CC7899	Xoan P.	Abogado	1997	6774434	15003							
Clave Cliente	ID Cliente	Nombre Cliente	Profesión	Año Vinculación	ID Hogar	Código Postal																														
8876744	CC7899	Juan P.	Abogado	1997	6774434	15003																														
Clave Cliente	ID Cliente	Nombre Cliente	Profesión	Año Vinculación	ID Hogar	Código Postal																														
8876744	CC7899	Xoan P.	Abogado	1997	6774434	15003																														
<b>SCD-2</b>																																				
Inserción de un nuevo registro	<table border="1"> <thead> <tr> <th>Clave Cliente</th><th>ID Cliente</th><th>Nombre Cliente</th><th>Profesión</th><th>Año Vinculación</th><th>ID Hogar</th><th>Código Postal</th></tr> </thead> <tbody> <tr> <td>8876744</td><td>CC7899</td><td>Juan P.</td><td>Abogado</td><td>1997</td><td>6774434</td><td>15003</td></tr> </tbody> </table> <table border="1"> <thead> <tr> <th>Clave Cliente</th><th>ID Cliente</th><th>Nombre Cliente</th><th>Profesión</th><th>Año Vinculación</th><th>ID Hogar</th><th>Código Postal</th></tr> </thead> <tbody> <tr> <td>8876744</td><td>CC7899</td><td>Xoan P.</td><td>Abogado</td><td>1997</td><td>6774434</td><td>15003</td></tr> <tr> <td>8892313</td><td>CC7899</td><td>Xoan P.</td><td>Abogado</td><td>1997</td><td>6774434</td><td>28034</td></tr> </tbody> </table>	Clave Cliente	ID Cliente	Nombre Cliente	Profesión	Año Vinculación	ID Hogar	Código Postal	8876744	CC7899	Juan P.	Abogado	1997	6774434	15003	Clave Cliente	ID Cliente	Nombre Cliente	Profesión	Año Vinculación	ID Hogar	Código Postal	8876744	CC7899	Xoan P.	Abogado	1997	6774434	15003	8892313	CC7899	Xoan P.	Abogado	1997	6774434	28034
Clave Cliente	ID Cliente	Nombre Cliente	Profesión	Año Vinculación	ID Hogar	Código Postal																														
8876744	CC7899	Juan P.	Abogado	1997	6774434	15003																														
Clave Cliente	ID Cliente	Nombre Cliente	Profesión	Año Vinculación	ID Hogar	Código Postal																														
8876744	CC7899	Xoan P.	Abogado	1997	6774434	15003																														
8892313	CC7899	Xoan P.	Abogado	1997	6774434	28034																														
<b>SCD-3</b>																																				
Adición de un nuevo atributo	<table border="1"> <thead> <tr> <th>Clave Cliente</th><th>ID Cliente</th><th>Nombre Cliente</th><th>Profesión</th><th>Año Vinculación</th><th>ID Hogar</th><th>Código Postal</th></tr> </thead> <tbody> <tr> <td>8876744</td><td>CC7899</td><td>Xoan P.</td><td>Abogado</td><td>1997</td><td>6774434</td><td>15003</td></tr> </tbody> </table> <table border="1"> <thead> <tr> <th>Clave Cliente</th><th>ID Cliente</th><th>Nombre Cliente</th><th>Profesión</th><th>...</th><th>CP Inicial</th><th>CP Actual</th></tr> </thead> <tbody> <tr> <td>8876744</td><td>CC7899</td><td>Xoan P.</td><td>Abogado</td><td>...</td><td>15003</td><td>28034</td></tr> </tbody> </table>	Clave Cliente	ID Cliente	Nombre Cliente	Profesión	Año Vinculación	ID Hogar	Código Postal	8876744	CC7899	Xoan P.	Abogado	1997	6774434	15003	Clave Cliente	ID Cliente	Nombre Cliente	Profesión	...	CP Inicial	CP Actual	8876744	CC7899	Xoan P.	Abogado	...	15003	28034							
Clave Cliente	ID Cliente	Nombre Cliente	Profesión	Año Vinculación	ID Hogar	Código Postal																														
8876744	CC7899	Xoan P.	Abogado	1997	6774434	15003																														
Clave Cliente	ID Cliente	Nombre Cliente	Profesión	...	CP Inicial	CP Actual																														
8876744	CC7899	Xoan P.	Abogado	...	15003	28034																														

Tabla 4-4. Estrategias SCD para la gestión de cambios.

La Tabla 4-4 muestra un ejemplo de cada una de estas estrategias. El punto de partida es un registro, un cliente en este caso, con una serie de atributos. La primera vez que el cliente aparece en la dimensión se le asigna una clave subrogada. Es decir, tiene su ID de cliente, que es la clave natural y perpetua, pero el sistema de ETL le asigna esta clave primaria artificial. A partir de ese momento, todas las compras que realice quedarán reflejadas en la tabla de hechos con esa clave. Con el paso del tiempo los atributos del cliente cambiaran, pudiéndose dar los siguientes escenarios:

- **SCD-1: actualización.** El primer ejemplo es cuando no nos interesa preservar la historia. El cliente normaliza su nombre, y distinguir en las consultas e informes entre el antes y el después no nos aporta nada. En esta estrategia simplemente sobrescribimos el valor del atributo, y es típica cuando se trata de corregir errores en la introducción de los datos. Ahora bien, si hubiera algún informe que agregara la información por nombre del cliente (cosa extraña), habría que rehacerlo.
- **SCD-2: nuevo registro.** En este caso sí nos interesa mantener el antes y el después. El cliente cambia de código postal en un momento del tiempo y queremos diferenciar este hecho en las consultas. La manera de hacerlo es insertar un nuevo registro en la tabla de dimensión donde al cliente se le asigna una nueva clave subrogada. El resto de sus atributos son los mismos, excepto el código postal, que tiene el nuevo valor. A partir de ese momento, todas su compras aparecerán en la tabla de hechos con esa nueva clave artificial. De esta manera, no es necesario rehacer ningún informe existente y siempre podremos recuperar todo el historial del cliente accediendo por la clave natural<sup>100</sup>. En esta estrategia, el sistema de ETL tiene que encargarse del mantenimiento y generación de las nuevas claves, llevando un control también de las fechas de efectividad y expiración de cada una<sup>101</sup>. El principal inconveniente es que la tabla de dimensión, pese a no modificar su estructura, irá aumentando en tamaño.
- **SCD-3: nuevo atributo.** Hay situaciones en las que nos puede interesar mantener en el mismo registro tanto el valor actual del atributo como el anterior. En el tipo SCD-2 es difícil relacionar el antes y el después. Por ejemplo, nos puede interesar analizar como habrían sido las ventas por código postal si el cliente no se hubiese cambiado de domicilio (asumimos que se trata de un cliente muy importante). Esta estrategia nos permite hacerlo, aunque implica modificar la estructura de la tabla. Además, no es viable almacenar todo el histórico de posibles cambios, siendo razonable limitarnos al valor actual y al inmediatamente anterior. En este caso también será necesario reconstruir cualquier agregado o informe que contenga esos atributos que han cambiado.

---

<sup>100</sup> No hay que perder de vista que el propósito de las claves subrogadas nunca es el acceso a los datos (de hecho nunca se presentan al usuario final). El punto de entrada en las consultas es, entre otros atributos, las claves naturales.

<sup>101</sup> Esto es debido a que podemos tener la necesidad de cargar registros antiguos en la tabla de hechos, por lo que tenemos que saber que clave subrogadas les corresponde.

Existen documentados hasta ocho tipos de estrategias SCD (incluyendo el tipo SCD-0)<sup>102</sup>. La elección de una u otra para cada tipo de atributo en una dimensión debe estar siempre marcada por los requerimientos de negocio. Como decíamos al inicio de este apartado, los tipos SCD son válidos mientras el cambio en los atributos sea ocasional. En caso contrario tendríamos que emplear otro tipo de estrategias, como la segregación de atributos o las **minidimensiones**<sup>103</sup>.

La modelización multidimensional, así como los patrones de consulta que soporta, no es sencilla en **gestores NoSQL**<sup>104</sup>. Entre todos ellos, el basado en columnas sería el más próximo, ya que consiste también en tablas. Además, el almacenamiento conjunto de columnas es muy beneficioso para las operaciones de agregación. Sin embargo, estas bases de datos resuelven las relaciones entre tablas a nivel de aplicación, por lo que no queda más remedio que desnormalizar todavía más el modelo, consolidando todos los datos del esquema en estrella en una única tabla. La idea sería poner todas las métricas de la tabla de hechos en una familia de columnas, y cada dimensión en su propia familia, de forma que cada atributo tendría su propia columna. Las capacidades de crecimiento de estos gestores, mediante una división horizontal de los datos, permitirían manejar grandes volúmenes, si bien el tamaño de la tabla aumentaría enormemente al llevar la desnormalización al extremo. Además, la gestión de SCD sería, a su vez, complicada con una única tabla. Por otro lado, la segregación en familias de columnas resultaría realmente efectiva cuando las consultas involucrasen más de un atributo de una dimensión, cuando la tendencia es a incluir solo uno. Por último, la comparación de los rendimientos alcanzados en las pruebas no indica una superioridad de estos gestores frente a los relacionales para estas cargas de trabajo<sup>105</sup>.

### 4.3 TECNOLOGÍAS PARA EL TRATAMIENTO DE DATOS

Vamos a ver las tecnologías de las que disponemos para el tratamiento de los datos. Estas no se limitarán al aprovisionamiento de nuestros sistemas de información, sino que serán empleadas siempre que exista el requerimiento de procesar datos en bloque.

102 [https://en.wikipedia.org/wiki/Slowly\\_changing\\_dimension](https://en.wikipedia.org/wiki/Slowly_changing_dimension)

103 <https://dwbi.org/pages/22>

104 La excepción serían las **bases de datos MOLAP** (*Multidimensional OLAP*) que, con un almacenamiento matricial, están especializadas precisamente en representar este tipo de estructuras. Se utilizan principalmente en tareas de simulación, planificación y presupuestación. Sin embargo, no se caracterizan ni por su capacidad de crecimiento ni por el volumen de datos o el nivel de concurrencia de usuarios soportado, por lo que no son habituales en entornos de *Big Data*. Oracle Essbase o IBM TM1 son dos ejemplos.

105 <https://www.diva-portal.org/smash/get/diva2:1222829/FULLTEXT01.pdf>

### 4.3.1 Apache Hadoop

En el Apartado 3.2.1 hablamos del sistema de archivos distribuido de Apache Hadoop, HDFS, e introdujimos sus otros dos componentes, enfocados estos al procesamiento de datos que residen en el clúster. La pieza central (Figura 3-4) es YARN (*Yet Another Resource Negotiator*), un marco de trabajo para la gestión de recursos y planificación de tareas en el clúster. Es decir, Hadoop nos proporciona un sistema de archivos distribuido en nodos de datos, y un entorno para ejecutar aplicaciones en nodos de cómputo. La idea es que ambos nodos coincidan y así aprovechar la localidad del dato.

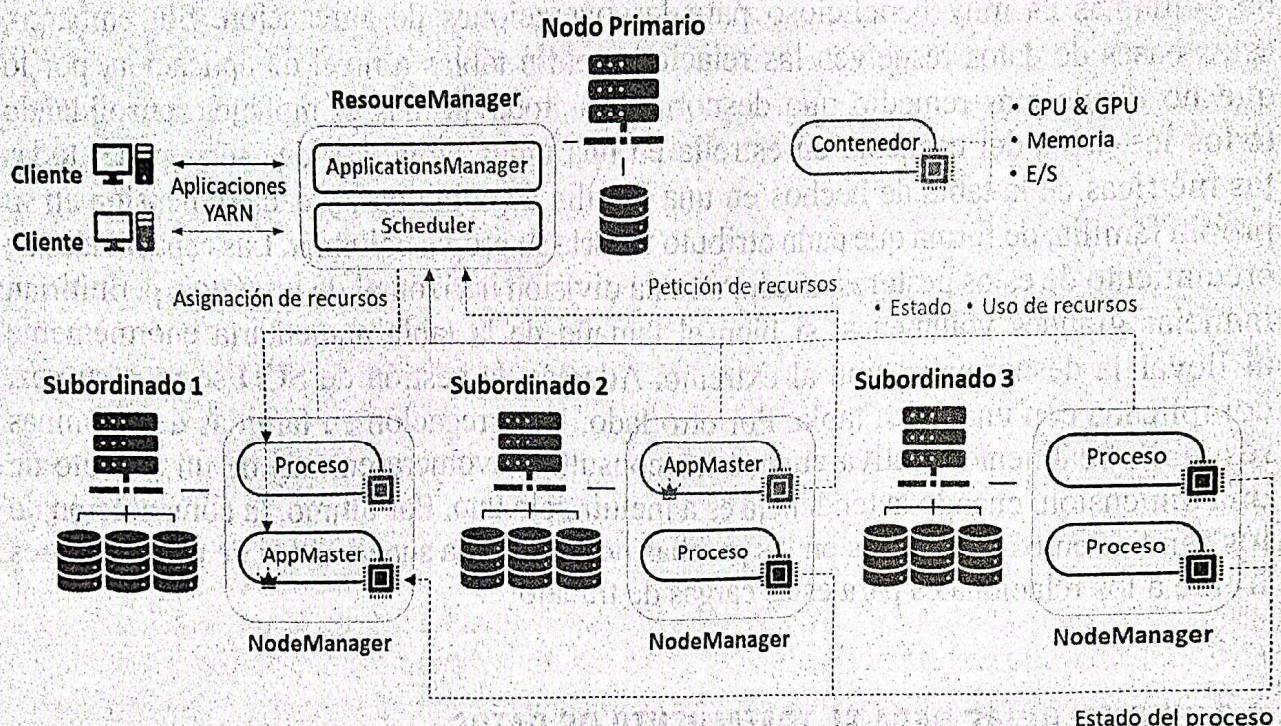


Figura 4-6. Arquitectura de YARN

YARN expone los recursos del clúster para su manipulación, y estos pueden ser utilizados mediante algún sistema de procesamiento que implemente un modelo de programación compatible. Dentro del propio Hadoop, **MapReduce** es el componente con el que generar aplicaciones YARN, pero puede ser reemplazado por otros sistemas externos, como **Apache Tez** o **Apache Spark**. MapReduce está más orientado a aplicaciones por lotes, mientras que Tez y Spark lo están a aplicaciones interactivas y en memoria, respectivamente. De la misma forma que podemos trabajar en Hadoop con otros sistemas de almacenamiento diferentes a HDFS, podemos utilizar otras alternativas a MapReduce para el cómputo.

La Figura 4-6 esquematiza los componentes y la forma en la que se despliega una aplicación YARN. Podemos comentar los siguientes:

- Existen dos tipos de nodos en un cluster de YARN: el **nodo primario**, encargado de gestionar de forma global los recursos, y donde se ejecuta el **ResourceManager** como proceso, y una serie de **nodos subordinados**, ejecutándose en cada uno un proceso denominado **NodeManager**.

- ▶ El ResourceManager tiene a su vez dos componentes: el **Scheduler**, que gestiona y asigna los recursos de cómputo del clúster en función de su estado y del requerimiento de las aplicaciones, y el **ApplicationsManager**, encargado de recibir las peticiones de ejecución de aplicaciones, iniciarlas y controlar su estado y desenlace.
- ▶ Las aplicaciones YARN pueden ser sencillas, de una sola tarea, o bien de naturaleza distribuida, constituidas por un conjunto de ellas. En cualquier caso, toda tarea en YARN se ejecuta de forma aislada dentro de un **contenedor**, que tiene sus propios recursos de CPU, memoria, disco, etc.
- ▶ Cuando un cliente, a través del API de YARN, envía al nodo primario una aplicación para su ejecución, el ResourceManager ubica y lanza en un nodo subordinado un primer contenedor, denominado **ApplicationMaster**. Si la aplicación es multitarea, entonces el ApplicationMaster solicita al ResourceManager la creación de contenedores adicionales para la ejecución de los procesos. De lo contrario, la aplicación se ejecutará en el propio ApplicationMaster, devolviendo el resultado al cliente. Es responsabilidad del ApplicationsManager la monitorización de cada ApplicationMaster ejecutándose en el clúster, reiniciándolo en caso de fallo del contenedor.
- ▶ Los contenedores de procesos de la aplicación son distribuidos por los nodos subordinados del clúster de acuerdo a las especificaciones de la aplicación y a la disponibilidad de recursos. La aplicación puede requerir que ciertas tareas se ejecuten en nodos con ciertas características, como la disponibilidad de GPU, o que pertenezcan a un mismo bastidor del clúster.
- ▶ El ApplicationMaster se encarga de comunicarse con el resto de los contenedores de la aplicación, coordinando su progreso, reiniciándolos en casos de fallo, y actuando como intermediario con el ResourceManager para la solicitud y liberación de recursos.

En un clúster pequeño, no superior a 10 nodos, lo habitual es que el nodo primario de YARN conviva en el mismo nodo con el NameNode de HDFS, habida cuenta que existe un NameNode secundario. En un clúster de tamaño mayor es recomendable separarlos. Respecto a los nodos subordinados de YARN, lo recomendable es instalar un NodeManager en cada DataNode de HDFS, para así aprovechar mejor los recursos de estos servidores y poder planificar las tareas en los nodos donde los datos ya están presentes<sup>106</sup>, aunque hay excepciones a esta regla.

106 Cada contenedor se encarga de obtener los datos que necesita comunicándose con el cliente HDFS del nodo en el que se está ejecutando.

### 4.3.2 Aplicaciones MapReduce

**MapReduce** es un modelo genérico de programación distribuida y en paralelo, siendo su implementación en Hadoop una de las más populares. Si bien su utilización ha ido decayendo en los últimos años en favor de otras opciones, como Tez o Spark, es conveniente conocer sus características y ver sus posibles aplicaciones.

Una tarea MapReduce (*job*) está enfocada al procesamiento por lotes, y se compone de unos datos de entrada que queremos procesar, una función de transformación (*map*), otra de agregación (*reduce*)<sup>107</sup> y unos parámetros de configuración de la tarea.

La Figura 4-7 muestra el flujo de datos de una consulta que podemos plantear en términos de una aplicación MapReduce. Disponemos de valores diarios de calidad del aire (concentración de SO<sub>2</sub> en µg/m<sup>3</sup>) medidos por múltiples sensores esparcidos por una determinada área, y recogidos a lo largo de varias décadas. El objetivo es obtener el valor promedio de calidad para cada día durante todo ese periodo. Para ello, la aplicación se compone de las siguientes etapas:

- ▶ **Entrada.** Los datos se presentan como un conjunto de archivos CSV residentes en HDFS, con un archivo por cada año de mediciones. Cada registro en un archivo consta de la fecha, el identificador del sensor y el valor de la calidad del aire.

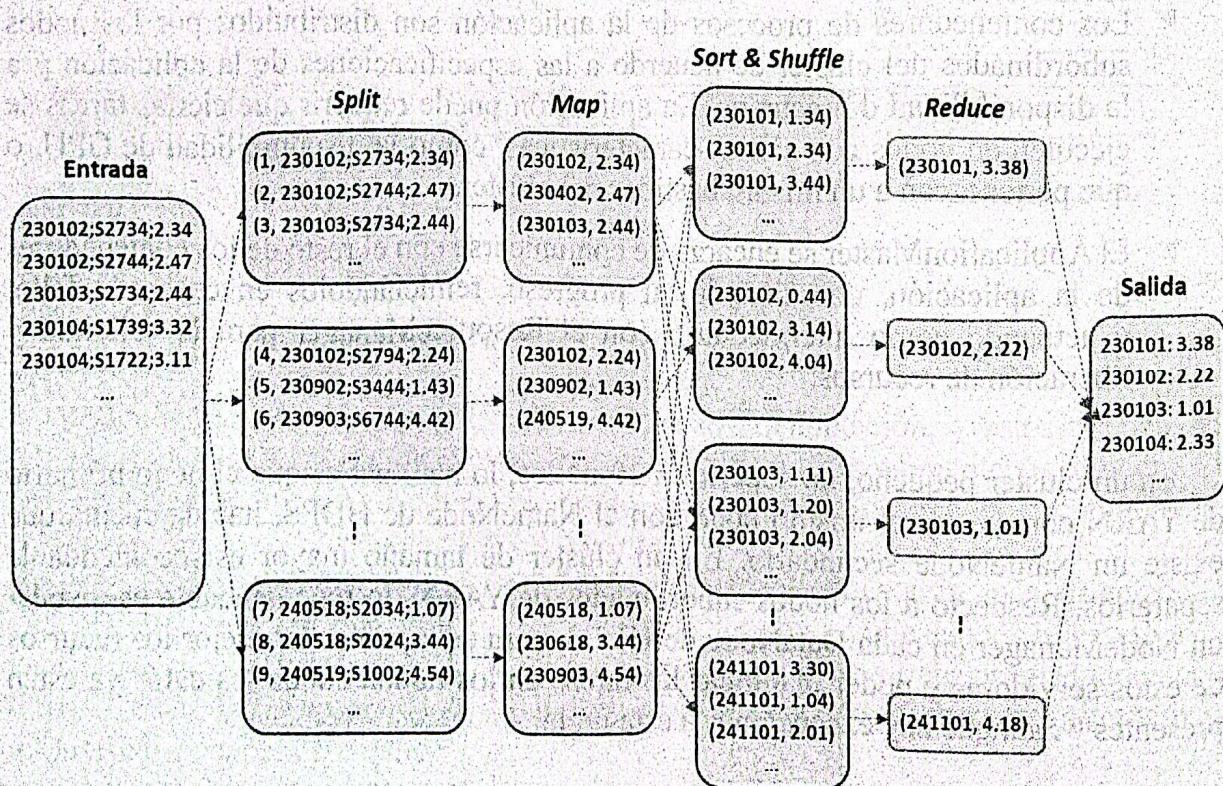


Figura 4-7. Etapas conceptuales de una aplicación MapReduce.

<sup>107</sup> Las funciones de cálculo y agregación pueden escribirse en distintos lenguajes de programación, como Java, Python o C++, entre otros.

- ▶ **División (*split*)**. Con el fin de parallelizar la ejecución de la tarea y diseminarla por los distintos nodos del clúster, los datos de entrada son fraccionados en divisiones, cuyo tamaño se suele hacer coincidir con el tamaño del bloque en HDFS. Cada etapa en MapReduce recibe y produce datos en forma de clave-valor, por lo que cada división contiene pares de este tipo, siendo la clave el número de registro y el valor el propio registro. En términos de YARN, el despliegue de la aplicación implica el de un contenedor con el ApplicationMaster, que se encarga de su gestión.
- ▶ **Transformación (*map*)**. El ApplicationMaster se encarga de crear una tarea de transformación independiente para cada división de datos, creando contenedores que ejecutan la función en los diferentes nodos, de acuerdo con el planificador de YARN. Si el tamaño de cada división coincide con el del bloque en HDFS, entonces es posible la localidad del dato; es decir, es factible ubicar y ejecutar un contenedor en el mismo nodo que contiene una división<sup>108</sup>. En el ejemplo, la función de transformación proporcionada se encarga de extraer la fecha y la calidad de cada registro, descartando el resto del contenido<sup>109</sup>. Esta etapa de transformación producirá una serie de resultados intermedios, que se almacenan localmente en el nodo<sup>110</sup>.
- ▶ **Ordenación (*sort*)**. El resultado de aplicar la función de transformación sobre cada división tiene que ser ordenado por clave antes de ser enviado a la etapa de agregación. Esta etapa es realizada de forma automática por MapReduce. En nuestro caso, podemos garantizar que la salida de cada contenedor de transformación está ordenada por la fecha de la medición.
- ▶ **Agregación (*reduce*)**. Las salidas ordenadas de cada nodo son enviadas a un último nodo del clúster (aquí no hay posibilidad de localidad) donde se fusionan (*shuffle*) y aplica la función de agregación en su propio contenedor YARN<sup>111</sup>. Es decir, para cada fecha se combinan los distintos valores de calidad del aire y se calcula sobre ellos el valor promedio. Este resultado final sí es escrito en HDFS, con la consiguiente replicación entre nodos.

<sup>108</sup> Es posible pero no está asegurado, ya que todos los nodos que contienen réplicas de la división podrían estar, de acuerdo con el planificador, ejecutando otras tareas. En cualquier caso, la intención sería crear el contenedor lo más cerca posible, en el mismo bastidor, por ejemplo.

<sup>109</sup> La función de transformación puede contener mucha más lógica, como filtros, cálculos, conversiones, etc.

<sup>110</sup> Estos resultados intermedios no se almacenan en HDFS ya que la replicación automática es innecesaria y ralentizaría la ejecución: si una tarea de transformación falla el ApplicationMaster se encargará de volver a ejecutarla con otro contenedor.

<sup>111</sup> Hay aplicaciones en las que puede haber más de una tarea de agregación, o incluso ninguna, de forma que la ejecución terminaría al escribirse la salida de cada tarea de transformación a HDFS, desarrollándose todo el proceso en paralelo.

Un procesamiento en paralelo e independiente como el descrito se adapta muy bien a muchos tipos de aplicaciones donde hay que gestionar grandes volúmenes de datos de forma escalable, como el análisis de registros web, el comportamiento en redes sociales o el procesamiento de imágenes y textos.

### 4.3.3 Apache Spark

En la página web del proyecto, **Apache Spark**<sup>112</sup> es definido como un motor analítico unificado para el procesamiento de datos a gran escala. Puede ser instalado en un único nodo o en un clúster de servidores. En este último caso, el despliegue puede hacerse de forma autogestionada (*standalone mode*), o bien sobre un gestor de clústeres existentes, como **Hadoop YARN** o **Kubernetes**. La Figura 4-8 muestra los principales módulos de Spark.

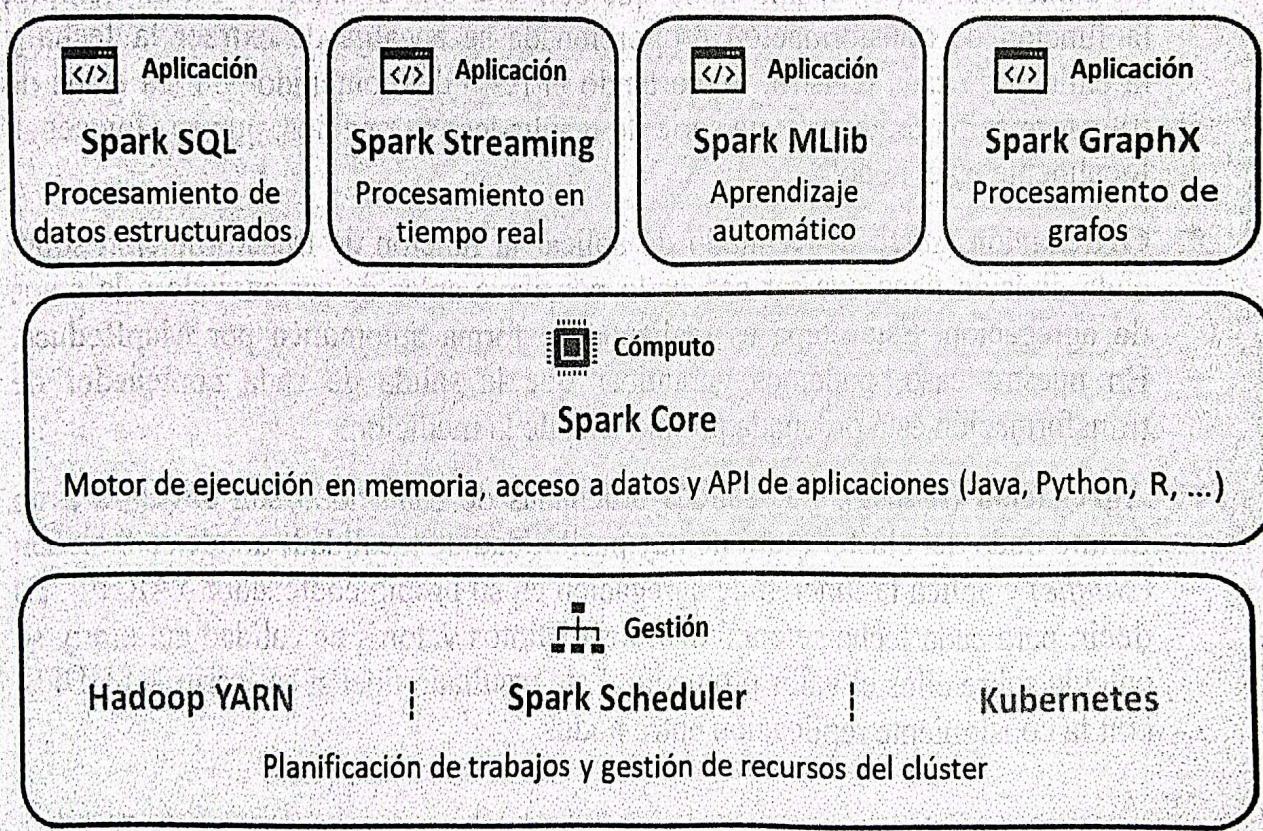


Figura 4-8. Módulos de Apache Spark.

Lo que diferencia principalmente a Spark son sus características de ejecución en memoria, concretamente su capacidad para almacenar en RAM los resultados intermedios, minimizando el acceso a disco durante todo el procesamiento. Esto lo hace extremadamente rápido, especialmente cuando se compara con Hadoop. Esta ejecución gira alrededor de la abstracción inicial de los datos en unas **estructuras resilientes y distribuidas (RDD, Resilient Distributed Dataset)**, que se reparten a lo largo de los nodos del clúster y pueden ser operadas en paralelo y restituidas en caso de fallo. Los programas de Spark manipulan estas estructuras en memoria mediante **operaciones**, que

se pueden dividir en **transformaciones** y **acciones**, ya sea directamente o a través de otras abstracciones de conveniencia, como los **DataFrames** y los **Datasets**. Spark es un motor totalmente desacoplado del almacenamiento, del que no proporciona mecanismo alguno. Por el contrario, exhibe una gran conectividad a todo tipo de bases de datos, sistemas de archivos y almacenes de objetos, de donde obtiene los datos para cargar sus estructuras en memoria y operar sobre ellas.

La Tabla 4-5 enumera y describe las funciones de cada uno de los módulos de Spark. **Spark GraphX** es un módulo que puede utilizarse de forma combinada con bases de datos de grafos, como Neo4J, en aquellas situaciones en las que más allá de la persistencia se requiere procesar de forma rápida grandes conjuntos de datos. **Spark SQL** lo comentaremos más adelante en el capítulo cuando abordemos los motores de consulta; **Spark Streaming** es un módulo para el procesamiento de eventos en tiempo real, mientras que **Spark MLlib** se utiliza en análisis predictivo.

Módulo	Funciones
<b>Spark Core</b>	Proporciona el motor de ejecución en memoria común al resto de módulos, encargándose de la gestión y manipulación de los RDD, así como del acceso y la persistencia de los datos en repositorios externos, la monitorización, planificación, etc.
<b>Spark SQL</b>	Se encarga de procesar datos estructurados, apoyándose en dos abstracciones sobre el RDD, que son el Dataset y el DataFrame, equivalente este último a una tabla relacional. Permite manipular estas estructuras a través de un API específico o de SQL, soportando conexiones a través de JDBC/ODBC.
<b>Spark Streaming</b>	Motor de procesamiento de flujos de datos basado en Spark SQL que trabaja con bajas latencias manipulando lotes ( <i>microbatch</i> ) o en continuo. Proporciona funciones de agregación temporal y manipulación de eventos sobre Datasets y DataFrames.
<b>Spark MLlib</b>	Conjunto de librerías para aprendizaje automático ( <i>machine learning</i> ) que trabajan sobre el DataFrame como estructura de datos, proporcionando múltiples algoritmos para tareas de predicción, clasificación o segmentación de datos.
<b>Spark GraphX</b>	Módulo para el procesamiento en paralelo de grafos que extiende el RDD para representar estructuras de nodos y arcos (RDG, <i>Resilient Distributed Graph</i> ), añadiendo operadores y funciones para su interrogación y transformación.

Tabla 4-5. Funciones de los módulos de Apache Spark.

Aunque hemos indicado que Spark se puede desplegar de forma autogestionada, son todavía muy frecuentes las instalaciones donde se ejecuta sobre YARN, aprovechando sinergias con HDFS para la persistencia de los datos. La Figura 4-9 muestra este despliegue, siendo equivalente sobre otros gestores.

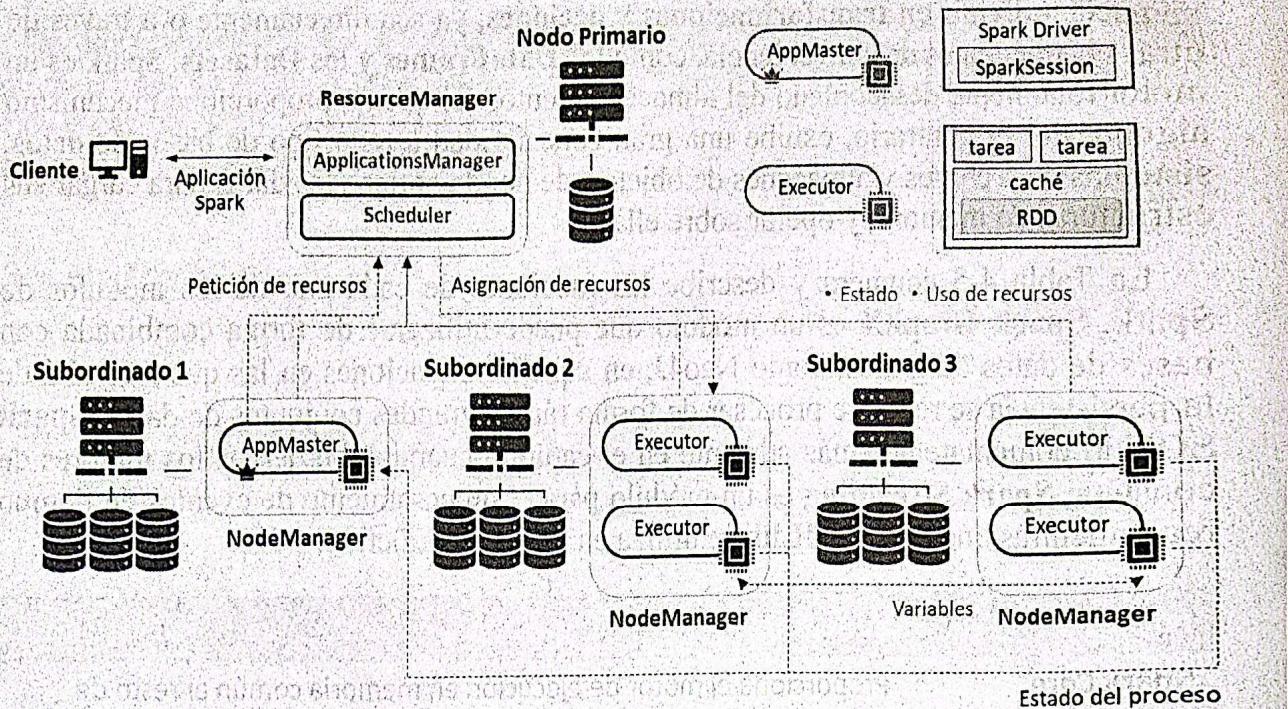


Figura 4-9. Despliegue de una aplicación Spark sobre YARN.

En este caso, el ApplicationMaster de YARN se compone de un programa, denominado **Spark Driver**, que se encarga de orquestar las operaciones en el clúster<sup>113</sup>. Entre otras tareas, este programa crea una instancia de **SparkSession**, que es el elemento que canaliza las peticiones de acceso y creación de estructuras de datos, incluyendo las operaciones con los distintos módulos. En cada nodo se generan, en forma de contenedor YARN, uno o varios **Spark Executor**, que se encargan de ejecutar tareas de Spark Core, manipulando las estructuras de datos en memoria en forma de RDD. Otra característica de Spark es el intercambio de variables entre tareas, incluso entre contenedores en diferentes nodos. De esta manera es posible compartir valores en memoria, como contadores o agregados, resultado de la aplicación de funciones que se ejecutan en paralelo.

Una de las características más relevantes de Spark es la forma en la que opera sobre los RDD. La Figura 4-10 esquematiza esta idea. Los datos son leídos desde HDFS<sup>114</sup>, aprovechando la localidad de los bloques en cada nodo, y cargados en la memoria caché de cada Spark Executor. A partir de ahí, y de acuerdo con la finalidad de la aplicación (por ejemplo, una consulta), cada partición del RDD es sometida a una serie de **transformaciones**, operando el conjunto en paralelo. Toda partición es **inmutable**, lo que quiere decir que una transformación resulta siempre en una nueva partición. Esta inmutabilidad permite que las transformaciones se ejecuten en paralelo, optimizando el

<sup>113</sup> Existe otra opción de despliegue en YARN en la que el Spark Driver no se crea en el clúster, sino que reside con la aplicación cliente. Este modo permite aplicaciones interactivas (*shell*), mientras que cuando es en el clúster solo se soportan procesos por lotes. El despliegue en el clúster es tolerante a fallos, ya que el ApplicationMaster será regenerado en caso de incidencia, por lo que es el modo preferido para aplicaciones en producción.

<sup>114</sup> La idea es similar cuando los orígenes residen en otro tipo de repositorio.

uso de la memoria y facilitando la recuperación ante fallos, ya que es posible reconstruir todo el linaje en un nodo alternativo del clúster.

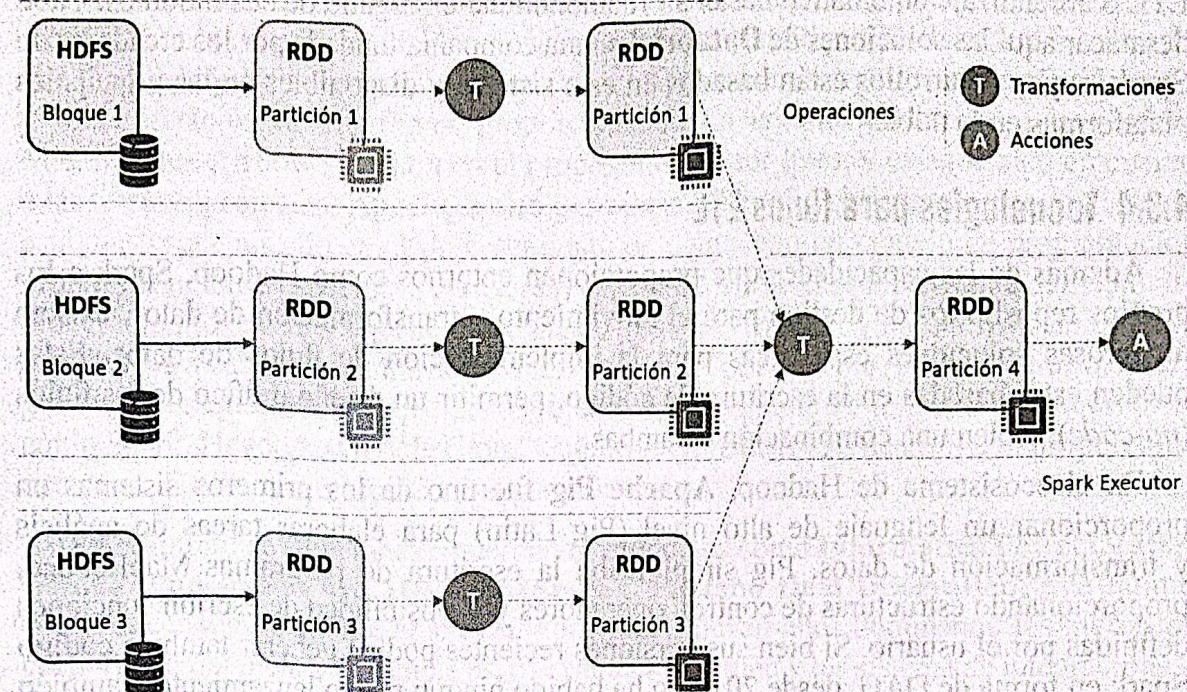


Figura 4-10. Ejecución de operaciones en paralelo sobre particiones RDD en Spark.

La librería de transformaciones que facilita Spark Core para RDD es muy amplia: aplicación de funciones, divisiones, filtros, muestreo, unión, intersección, agrupación, ordenación, etc. Spark aplica una **evaluación diferida** (*lazy evaluation*) en todas las transformaciones. Esto es, las trasformaciones se encadenan y solo se ejecutan cuando se invoca una acción, emitiendo esta un resultado. Esto permite combinar, en la medida de lo posible, distintas operaciones, optimizando el uso de memoria y reduciendo la cantidad de datos que se transfieren entre Spark Executors. Las **acciones** de Spark, por lo tanto, desencadenan un conjunto de transformaciones, devolviendo un resultado. Este puede ser un valor (conteo, valor máximo, mínimo, etc.) o una instrucción de persistencia del resultado en disco.

La ejecución de una acción genera lo que se denomina un **grafo acíclico dirigido (DAG, Directed Acyclic Graph)**, un concepto que se usa también en MapReduce y Tez. Un DAG viene a representar un plan de ejecución de las distintas transformaciones, siendo el Spark Driver el encargado de materializarlo entre los distintos nodos del clúster. Como su nombre indica, en este plan no existen transformaciones que generen ciclos, moviéndose los datos siempre del principio hacia el fin, marcado este por la acción desencadenante (Figura 4-10). Esta forma de procesamiento hace que Spark sea muy eficiente para distintos tipos de tareas, como la programación e implementación de flujos ETL, tanto por lotes como en tiempo real, el análisis y la exploración de datos mediante consultas interactivas, o el entrenamiento y el despliegue de modelos de minería de datos. Las opciones para abstraer el RDD mediante Datasets y DataFrames, más fáciles de manipular con lenguajes como Python o R, explican también la popularidad de este sistema.

Como comentamos en su momento, los servicios en la nube alrededor de Hadoop suelen llevar Spark integrado dentro del clúster. Además, muchas otras soluciones de ETL o aprendizaje automático basan su funcionalidad en alguno de sus módulos. Cabe desatar aquí las soluciones de **Databricks**, una compañía fundada por los creadores de Spark, cuyos desarrollos están basados en este sistema y disponibles en las principales plataformas en la nube.

#### 4.3.4 Tecnologías para flujos ETL

Además de las capacidades que proporcionan entornos como Hadoop, Spark o los propios repositorios de destino para el movimiento y transformación de datos, existen numerosas soluciones específicas para la implementación de flujos de datos. Estas pueden estar basadas en la escritura de código, permitir un diseño gráfico de los flujos (*no-code*), o bien una combinación de ambas.

En el ecosistema de Hadoop, **Apache Pig** fue uno de los primeros sistemas en proporcionar un lenguaje de alto nivel (Pig Latin) para elaborar tareas de análisis y transformación de datos. Pig simplificaba la escritura de programas MapReduce, proporcionando estructuras de control, operadores y la posibilidad de escribir funciones definidas por el usuario. Si bien sus versiones recientes podían generar también código Spark en forma de DAG, desde 2017 no ha habido ningún nuevo lanzamiento. También en la órbita de Hadoop cabe destacar **Apache Oozie**, si bien su función no es la transformación de datos como tal, sino la definición y gestión de flujos de trabajo que incluyen programas MapReduce, Spark, operaciones en HDFS, transformaciones Pig, programas externos, etc. Oozie proporciona elementos y estructuras de control de flujos y rutas alternativas en caso de fallos.

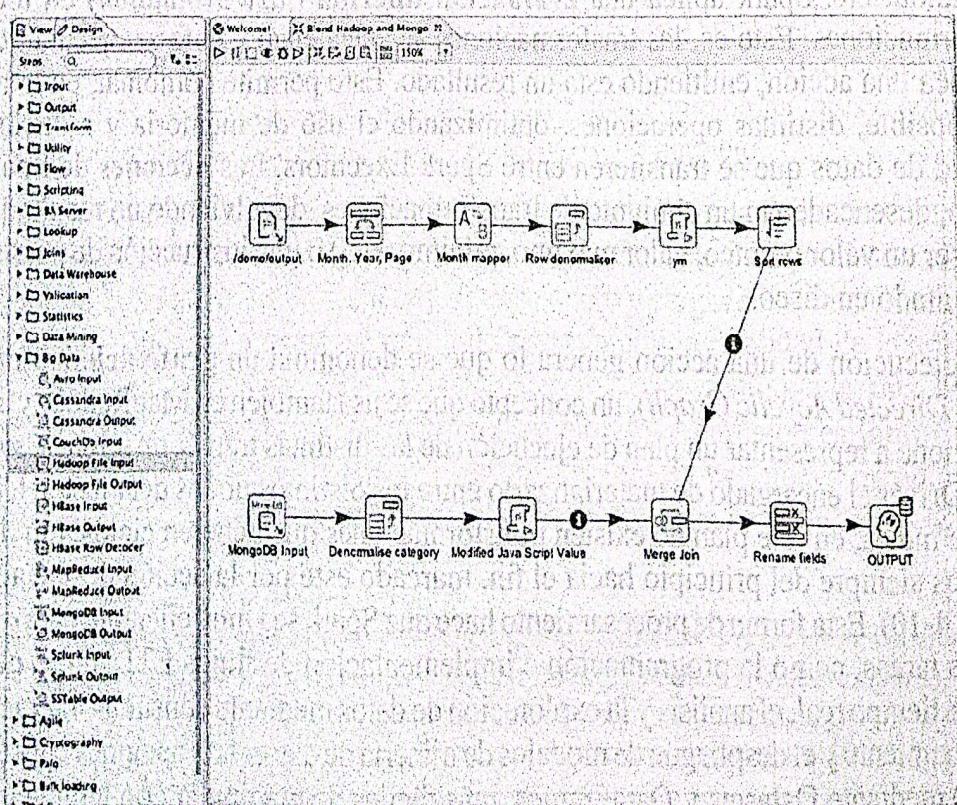


Figura 4-11. Flujo ETL en Pentaho Data Integration.

Dentro de los proyectos de código abierto hay dos soluciones a destacar. **Apache Airflow** es un sistema genérico para la construcción y ejecución de flujos de trabajo en forma de DAG, orquestando distintas tareas y dependencias entre ellas. Dispone de una amplia librería de operadores para definir las tareas, siendo posible la extensión de esta con funciones propias desarrolladas en Python. Airflow se encarga de toda la planificación y monitorización de los flujos, proporcionando un entorno gráfico para la gestión. Mientras que Airflow se centra en el procesamiento por lotes, y es agnóstico en cuanto a las tareas que ejecuta, **Apache Beam** permite también el trabajo en tiempo real. Beam está más especializado en el procesamiento de datos, con un modelo de programación propio y funciones específicas para tareas de ETL. Airflow es independiente del motor de procesamiento donde se ejecutan las tareas, pudiendo integrar y monitorizar un proceso que se lleva a cabo en un entorno de Spark separado, por ejemplo. Por el contrario, Beam genera flujos que deben desplegarse y ejecutarse en alguno de los motores soportados, como Spark, Hadoop o Google Cloud Dataflow, entre otros. Es decir, Beam proporciona una nivel de abstracción sobre estas plataformas.

En el ámbito de soluciones comerciales, productos como **Informatica PowerCenter**, **Oracle Data Integrator**, **IBM DataStage** o **Pentaho Data Integration**, este último con una versión de código abierto (Figura 4-11), llevan años dominando los mercados corporativos, si bien la tendencia es, nuevamente, la nube y los sistemas de integración de datos como servicio. Amazon dispone de dos soluciones para este fin, **AWS Glue** y **AWS Data Pipeline**. La primera es autogestionada (*serverless*) y se apoya en Spark para la ejecución, mientras que en la segunda se tiene una mayor control sobre la infraestructura, basada en servidores virtuales e instancias de AWS EMR, siendo más flexible y con más opciones. Google por su parte ofrece **GCP Dataflow** y **GCP Composer**. Este último es un servicio gestionado de Apache Airflow, mientras que el primero lo es de Apache Beam<sup>115</sup>. Finalmente, Microsoft basa su oferta en **Azure Data Factory**, que destaca por su amplia librería de conectores a fuentes de datos propias y externas, y sus funciones de transformación y control de flujos.

## 4.4 MOTORES DE CONSULTA DISTRIBUIDOS

Los lenguajes de interrogación a bases de datos son de naturaleza **declarativa**. Como contraposición a los lenguajes **imperativos**, como Java, C o Python, donde el código indica paso a paso las etapas que hay que seguir para obtener el resultado deseado, en un lenguaje declarativo se define el objetivo a conseguir, delegando la implementación y ejecución de los pasos concretos en un compilador. Como se suele decir, en un lenguaje declarativo se define el qué, pero no el cómo. Sin ser un lenguaje de programación, MapReduce se aproxima a un modelo imperativo, ya que empleamos lenguajes como Java para desarrollar las funciones de transformación y agregación. En el caso de Spark, aunque también utilizamos este tipo de lenguajes, la capa que se encarga de procesar

<sup>115</sup> De hecho, Apache Beam es el resultado de la donación del código de GCP Dataflow.

los RDD y generar el DAG es declarativa: nosotros no indicamos como debe ser el DAG, sino que este se crea como parte de un plan de acceso, optimizado de acuerdo a la estructura de los datos en ese momento, la complejidad del problema y los recursos disponibles en el clúster.

SQL es el gran ejemplo de los lenguajes declarativos. De forma equivalente a como lo hace Spark, cuando lanzamos una consulta, el compilador de la base de datos relacional se encarga de establecer el plan de acceso, teniendo en cuenta la posible existencia de índices que aceleren la consulta, las estadísticas de las tablas involucradas y la colocación de los datos, si el sistema está distribuido. Además, este plan de acceso es almacenado en memoria con el fin de poder ser reutilizado en futuras consultas de características similares. Los motores de consulta distribuidos siguen esta aproximación, pero desacoplando el almacenamiento: en lugar de reunir los datos bajo un único gestor, estos sistemas son capaces de interpretar comandos SQL y acceder a una variedad de orígenes y formatos, resolviendo distintas operaciones de manipulación y consulta. La diferencia con una base de datos SQL es, en ocasiones, sutil, ya que muchas de estas permiten trabajar también con tablas basadas en archivos que referencia datos en sistemas externos.

La existencia o no de un almacenamiento propio y dedicado, más allá de metadatos o de ciertas estructuras de agregación para optimizar las consultas, es lo que marca la diferencia y, al mismo tiempo, la aportación de los motores de consulta. En este sentido, una característica a remarcar es el soporte de **consultas federadas**, base de la **virtualización de datos**. Es decir, la posibilidad de crear una base de datos con tablas internas o externas que apuntan a distintos orígenes, permite realizar consultas que combinan datos de fuentes heterogéneas.

#### 4.4.1 Apache Hive

En base a estas características, la referencia en cuanto a motores de consulta es **Apache Hive**<sup>116</sup>. En su página web, Hive se presenta como un sistema de *data warehouse* basado en Hadoop<sup>117</sup>, que permite la consulta y el análisis de grandes volúmenes de datos utilizando un dialecto de SQL, denominado **HiveQL**, y un lenguaje procedimental, **Hive HPL/SQL**, para la creación de procedimientos almacenados, funciones o procesos ETL. Hive puede convertir la ejecución de sentencias SQL a tareas MapReduce, Tez o Spark, accediendo a datos ubicados en HDFS y otros repositorios, y soportando distintos formatos de archivo.

116 <https://hive.apache.org/>

117 Aunque requiere una serie de librerías, es posible llegar prescindir de Hadoop para la ejecución de Hive, desplegándose sobre Spark de forma autogestionada.

En producción es recomendable instalar los servicios de Hive en uno o varios nodos dedicados. El número de estos dependerá de la carga de trabajo esperada, pero su multiplicación permitirá segregar diferentes aplicaciones en función de sus necesidades (separando cargas ETL de consultas analíticas, por ejemplo) o implementar balanceo de carga y alta disponibilidad<sup>118</sup>.

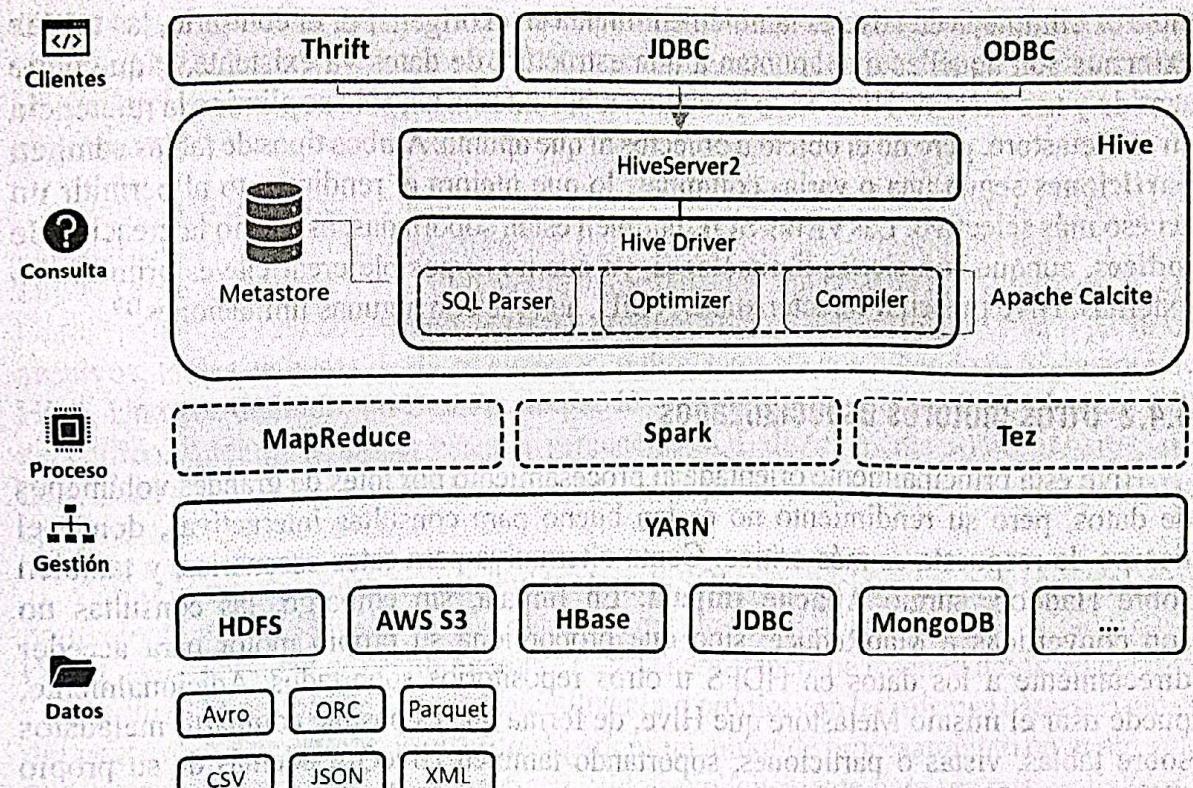


Figura 4-12. Arquitectura de Apache Hive.

En este escenario, Hive se componen de una parte servidora, **HiveServer2**, que se encarga de recibir y gestionar peticiones concurrentes de diversos clientes, gestionando a su vez la autenticación y la autorización de los accesos. HiveServer2 se comunica a su vez con un controlador, **Hive Driver**, cuya tarea es coordinar la elaboración del plan de acceso y su ejecución en forma de tarea MapReduce, Tez o Spark en el clúster de Hadoop. Un componente esencial de esta arquitectura (Figura 4-12) es el **Metastore**, una base de datos relacional que funciona como catálogo de las tablas, columnas, tipos de datos y otros metadatos del entorno.

<sup>118</sup> Apache ZooKeeper (<https://zookeeper.apache.org/>) es un sistema para la coordinación y sincronización de nodos en aplicaciones distribuidas. Es empleado por muchos otros proyectos de Apache, incluido Hadoop, Spark o Kafka para implementar alta disponibilidad, control de concurrencia o balanceo de carga, entre otras funcionalidades.

Como en otros gestores, Hive organiza los datos entorno al concepto de **base de datos**, entendido este como un espacio de nombres donde definir objetos. De igual manera, el concepto de tabla es central en el sistema. Hablamos de **tablas gestionadas** cuando estas son creadas y cargadas a través del propio Hive, como en un gestor relacional. Internamente estas tablas residirán en el origen de datos especificado (por defecto, en una serie de archivos ORC en un directorio reservado de HDFS), de forma que cuando la tabla es eliminada en Hive es también eliminada en el origen. Por el contrario, las **tablas externas** son aquellas que apuntan a una estructura de datos ya existente, y que tiene sentido y uso fuera de Hive. Si se borra una tabla externa tan solo se elimina la referencia en el Metastore, pero no el objeto u objetos al que apunta. Ambos tipos de tablas admiten **particiones** según una o varias columnas, lo que mejora el rendimiento al permitir un acceso más selectivo. Las **vistas SQL** también están soportadas, así como la creación de **índices**, aunque con ciertas limitaciones, ya que no es posible crear claves primarias o externas. Hive presenta características ACID, aunque con algunas limitaciones<sup>119</sup>.

#### 4.4.2 Otros motores especializados

Hive está principalmente orientada al procesamiento por lotes de grandes volúmenes de datos, pero su rendimiento no es tan bueno para consultas interactivas, donde el tiempo de respuesta es más crítico. Como alternativa para estos escenarios, y también sobre Hadoop, surgió **Apache Impala**. En Impala, sin embargo, las consultas no son convertidas a MapReduce, sino que proporciona su propio motor para acceder directamente a los datos en HDFS u otros repositorios soportados. Adicionalmente, puede usar el mismo Metastore que Hive, de forma que es posible compartir metadatos sobre tablas, vistas o particiones, soportando también HiveQL, además de su propio dialecto de SQL. Podríamos decir que Hive está más orientado a procesos ETL, mientras que Impala es más adecuado para análisis OLAP, donde la alta concurrencia y la baja latencia en los tiempos de respuesta son más importantes.

Aunque no es el caso de Impala, muchos motores de consulta están basados en **Apache Calcite**. Calcite no es un motor en sí, sino un marco de desarrollo (*framework*) para la construcción de sistemas de gestión de datos. Es totalmente agnóstico al almacenamiento y al procesado de los datos, situándose en medio de ambos, y permitiendo al desarrollador concretar el mecanismo de persistencia (RAM, JDBC, archivos, etc.), la definición de tabla o la implementación concreta de operadores de SQL (JOIN, GROUP BY, etc.). Por su parte, Calcite proporciona un intérprete de SQL (*parser*), un optimizador de consultas y generador de planes de acceso, que puede ser modificado mediante reglas, un conjunto de operadores, que también puede ser extendido, y un API y un controlador JDBC para la conexión. En definitiva, Calcite puede ser empleado para construir una base de datos SQL o un motor de consulta.

119 <https://cwiki.apache.org/confluence/display/Hive/Hive+Transactions>

Hive reemplazó a partir de la versión 0.14.0<sup>120</sup> el núcleo de su controlador con Calcite. Otro motor de consulta en el ámbito de Hadoop que está basado en Calcite es **Apache Phoenix**. Phoenix tiene un foco más operacional y está orientado a cargas OLTP, soportando transacciones ACID. Funciona añadiendo una capa SQL a HBase, en la que se apoya para el almacenamiento, no soportando la definición de tablas externas. **Apache Drill** es otro ejemplo de motor basado en Calcite, con una orientación más analítica y distribuida. El punto a destacar de Drill es su gran capacidad de federación de orígenes NoSQL, representando los datos mediante un modelo basado en JSON, lo que le proporciona una gran flexibilidad a la hora de interrogar datos sin esquema o con un esquema variable.

Como veíamos anteriormente en el capítulo, **Spark SQL** es el módulo de Spark para el procesamiento de datos estructurados. La principal estructura empleada para ello es el **DataFrame**, que viene a ser equivalente a una tabla en una base de datos relacional. Un DataFrame puede ser construido de forma programática, o bien se puede crear a partir de archivos, tablas en bases de datos externas, o directamente consultando el Metastore de Hive. Una vez definido, un DataFrame puede ser consultado empleando SQL o un API específico. Además de poder ejecutar sentencias SQL directamente sobre ficheros, sin necesidad de pasar por un DataFrame, Spark SQL permite también la creación de vistas temporales, ligadas a la duración de una sesión.

Como motor de consulta distribuido, Spark SQL proporciona un servidor Thrift<sup>121</sup> equivalente a HiveServer2, lo que permite consultas SQL directas a través de JDBC/ODBC, sin necesidad de escribir código. Para la persistencia de los metadatos, es posible compartir el Metastore de Hive, o bien utilizar uno propio, con las mismas características del primero. No perdamos de vista que Spark es un motor de procesamiento en memoria, por lo que los DataFrames deben ser persistidos explícitamente para ser materializados.

Por último, fuera de la Fundación Apache, otros motores de consulta de código abierto relevantes son **Trino** y **Presto**. Ambos comparten arquitecturas y conceptos similares a los que hemos visto, soportando un gran número de orígenes de datos heterogéneos y orientados a consultas OLAP. En los servicios en la nube podemos encontrar soluciones como **AWS Athena** o **IBM Cloud Data Engine**.

#### 4.4.3 Apache Arrow

Cuando hablamos del acceso distribuido a los datos, especialmente cuando participan diferentes sistemas y se emplean distintos modelos y lenguajes de programación, un elemento importante a tener en cuenta es la gestión y el intercambio de datos en memoria.

120 2014 (la primera versión de Hive es de 2010).

121 Apache Thrift es un lenguaje y un protocolo para el desarrollo de servicios e interfaces con el fin de conectar aplicaciones.

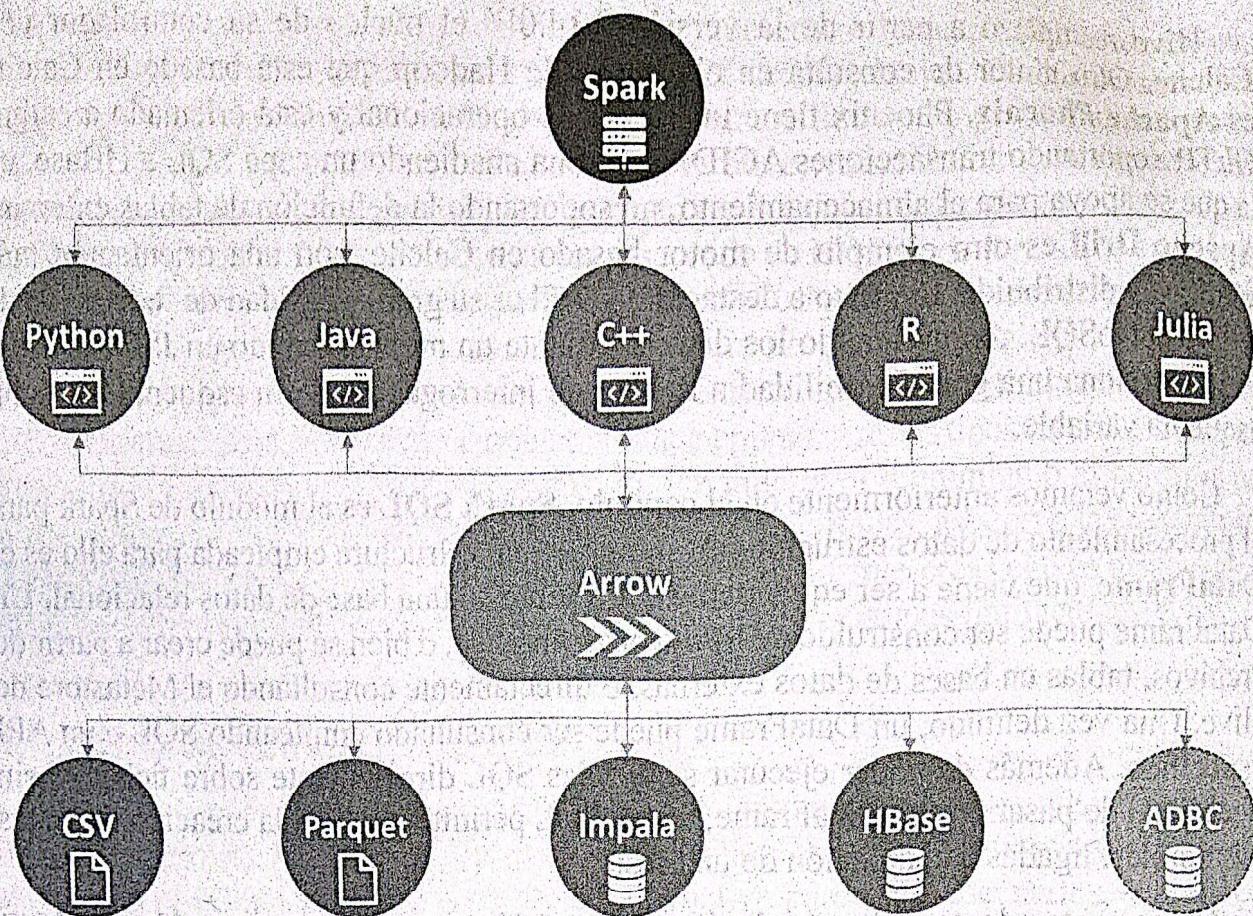


Figura 4-13. Intercambio de datos a través de Apache Arrow.

**Apache Arrow**<sup>122</sup> es un formato de representación en columnas de datos tabulares optimizado para operaciones analíticas, y que puede ser programado en distintos lenguajes. El objetivo de Arrow es estandarizar el movimiento, la manipulación y la transferencia de datos entre sistemas, sin necesidad de copias y conversiones entre los formatos internos de cada aplicación. Nuevamente, lo que perseguimos es desacoplar el procesamiento del formato, en este caso de la representación en memoria de los datos. Esto no solo significa un incremento de velocidad muy grande en operaciones de acceso debido a la optimización en columnas del formato, sino que permite la portabilidad directa de algoritmos entre lenguajes (Figura 4-13).

La web del proyecto Arrow resume las ventajas en el empleo de este formato alrededor de cuatro casos de uso<sup>123</sup>:

- ▀ **Lectura y escritura de archivos**, especialmente cuando la representación del dato es también en columnas, como Parquet<sup>124</sup>. En este caso, la identificación y conversión de metadatos es además directa, por lo que no es necesario volver a especificar el esquema.

122 <https://arrow.apache.org/>

123 [https://arrow.apache.org/use\\_cases/](https://arrow.apache.org/use_cases/)

124 Arrow dispone también de un formato de archivo, denominado Feather, para el almacenamiento directo de tablas.

- ▀ **Compartición de la memoria local**, lo que permite una manipulación eficiente de grandes volúmenes de datos a través de diferentes lenguajes.
- ▀ **Movimiento de datos en red**, con unos mecanismos de seriación y deseriación muy efectivos para grandes volúmenes de datos.
- ▀ **Estructura de datos en memoria para el análisis**, implementando motores de consulta distribuidos basados en el formato.

Aunque en fase de desarrollo, existe una especificación denominada **ADBC**, (*Arrow DataBase Connectivity*) enfocada a proporcionar un acceso estándar a distintas bases de datos, de forma equivalente a JDBC/ODBC.

## 4.5 RESUMEN DEL CAPÍTULO

En este capítulo nos hemos centrado en el procesamiento por lotes, contemplando como diseñar un modelo de datos que puede representar cambios, su aprovisionamiento y las opciones disponibles a la hora de consultarlos.

- ▀ Los **procesos ETL** son los encargados de obtener los datos de los sistemas origen, reconciliarlos, transformarlos y, finalmente, cargarlos en los repositorios de destino, donde se llevarán a cabo las tareas analíticas.
- ▀ El **modelado multidimensional** permite tanto una organización eficiente de los datos de cara a su consulta como la gestión de los cambios, soportando distintas estrategias para el mantenimiento de versiones.
- ▀ **Apache Hadoop** y **Apache Spark** son dos proyectos de código abierto para el tratamiento distribuido de grandes volúmenes de datos. Mientras el primero está más enfocado a tareas por lotes, el segundo es más efectivo para procesos interactivos, dadas sus capacidades de procesado en memoria sobre formatos de datos optimizados, como **Apache Arrow**.
- ▀ Los **sistemas de consulta distribuidos** se basan en desacoplar el almacenamiento del motor de acceso, permitiendo la manipulación mediante lenguaje SQL de estructuras de datos que residen en gestores dispersos y heterogéneos, sin necesidad de consolidarlos físicamente.

En el siguiente capítulo nos centraremos en el procesamiento en tiempo real, pero planteando también como podemos unificar ambos tipos de integración bajo un mismo sistema.