Capítulo 1

Sistemas de almacenamiento y procesado distribuido Introducción

Javier Esparza Peidro - jesparza@dsic.upv.es

Contenido

1.1. Introducción

Se estima que en el año 2020 cada usuario generó al menos 1.7 Mb por segundo. Con este ritmo de generación de datos es posible predecir que en el

2025 cada día se generará alrededor de 463 exabytes (10^3 petabytes). Sobre 2025 la totalidad de la información almacenada a nivel mundial llegará a los 175 billones de gigabytes. Esta estremecedora cifra pone de manifiesto que el almacenamiento de datos ha dejado de ser un problema secundario para pasar a ser uno de los mayores desafíos de la informática en el presente y un futuro cercano. En este contexto, el reto consiste en diseñar sistemas de almacenamiento capaces de albergar volúmenes ingentes de información, y herramientas y técnicas que permitan procesarla de manera eficiente.

1.2. Datos

Los requerimientos de datos de las aplicaciones actuales son mucho más exigentes que en tiempo pasados. Es habitual referirse a estos datos como *Big data*. *Big data* puede definirse como cualquier fuente de datos que posee al menos tres características (las 3 Vs):

- Volumen (volume): posee un volumen de datos muy grande, imposible de almacenar en una única máquina.

- Velocidad (velocity): los datos se generan con gran velocidad, a menudo en tiempo real. Esto implica que el volumen de información generada en un corto espacio de tiempo es muy grande.

- Variedad (variety): la naturaleza de los datos es muy variada, en cuanto a su procedencia y estructura interna. Existen datos estructurados (e.g. registros en una base de datos relacional, etc.), semi-estructurados (e.g. XML, JSON, etc.) o no estructurados (e.g. texto, imágenes, audio, etc.) .

Debido a estas tres características básicas, este volumen de datos es muy difícil de almacenar, manejar, procesar y analizar con técnicas tradicionales (i.e. bases de datos relacionales). Requiere coordinar diversos sistemas de almacenamiento, y utilizar nuevas técnicas y herramientas.

Además, cuando se habla de Big data, es habitual añadir algunas características adicionales (las 10 Vs):

- Veracidad (veracity): tiene que ver con la precisión, consistencia y corrección de los datos. Esta característica aglutina múltiples factores como la objetividad/subjetividad de los datos, la reputación/credibilidad de la fuente de datos, su disponibilidad, etc. y da una medida de la fiabilidad de los datos.

- Variabilidad (variability): tiene que ver con la varianza de propiedades esenciales de los datos, como su significado, inconsistencias, velocidad de adquisición, etc.
- Valor (value): determina cuán útiles son los datos para un propósito determinado.
- Validez (validity): está relacionado con la veracidad. Tiene que ver con la precisión y corrección de los datos en relación a un propósito determinado.
- Vulnerabilidad (vulnerability): tiene que ver con la seguridad de los datos.
- Volatilidad (volatility): está relacionado con el período de tiempo en el que los datos son válidos, antes de resultar obsoletos.
- Visualización (visualization): tiene que ver con la habilidad de presentar los resultados del análisis en un contexto gráfico que facilite su comprensión.

1.3. Sistemas de datos

Los sistemas que la mayor parte del tiempo están tratando con grandes volúmenes de datos, con datos de gran complejidad o con datos que cambian a gran velocidad (las 3 Vs) suelen denominarse *data-intensive* [Kleppmann, 2017], en contraposición con los sistemas *cpu-intensive*, en los que el factor limitante suele ser el consumo de CPU.

Los sistemas *data-intensive* han adquirido una complejidad tal que se construyen a partir de piezas básicas de diversa índole, como bases de datos, caches, índices de búsqueda, colas de mensajes, sistemas de procesamiento en batch y en streaming, etc.. Estas piezas gestionan los datos utilizando diversas estrategias y suministran diferentes garantías y mecanismos de acceso. Son comúnmente referidas como *sistemas de datos*.

En la figura 1.1 se ilustra un ejemplo de una aplicación data-intensive que integra una cache, una base de datos, un índice de búsqueda y una cola de mensajes. En este sistema las escrituras podrían propagarse a través de la cola de mensajes (ver figura 1.2a). En cambio, las búsquedas utilizarían el índice, para encontrar el identificador de los elementos que verifican la consulta, y después se recuperarían dichos elementos bien desde la cache, bien desde la base de datos (ver figura 1.2b). Todo el sistema se recubre con una API que desde el punto de vista de un observador externo podría comportarse como un nuevo sistema de datos.

En su definición más básica, un sistema de datos almacena datos y responde a preguntas en base a dichos datos. Dependiendo del propósito del sistema de

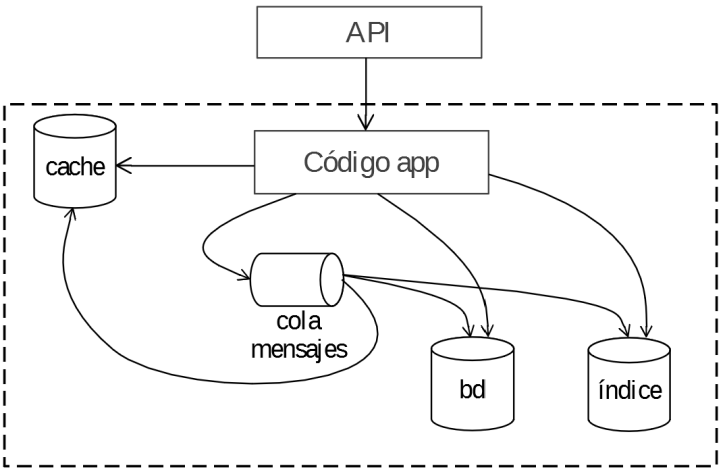


Figura 1.1: Ejemplo de aplicación data-intensive

datos, resultará prioritario optimizar las operaciones de escritura, de lectura o llegar a un compromiso entre ambas.

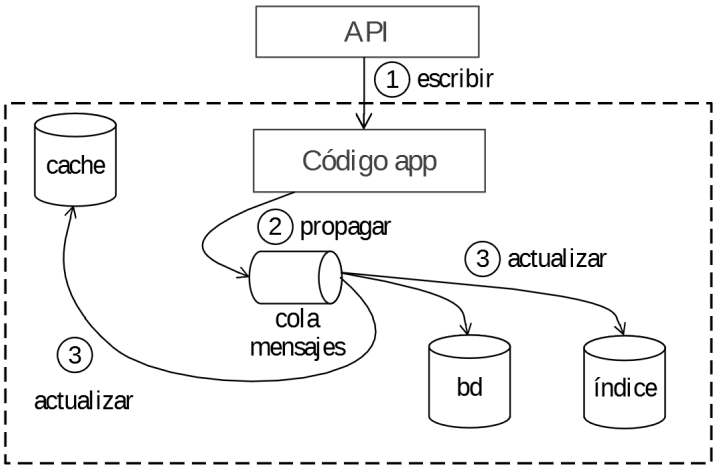
Tradicionalmente, la literatura ha clasificado los sistemas de datos en dos grandes categorías:

- Sistemas de tipo *Online Transaction Processing* (OLTP): son sistemas interactivos que efectúan operaciones "cortas" de lectura/escritura, denominadas transacciones, sobre el almacén de datos. En estos sistemas, el almacén está optimizado para devolver los resultados inmediatamente, y ofrece una alta productividad (i.e. un alto número de transacciones por segundo).

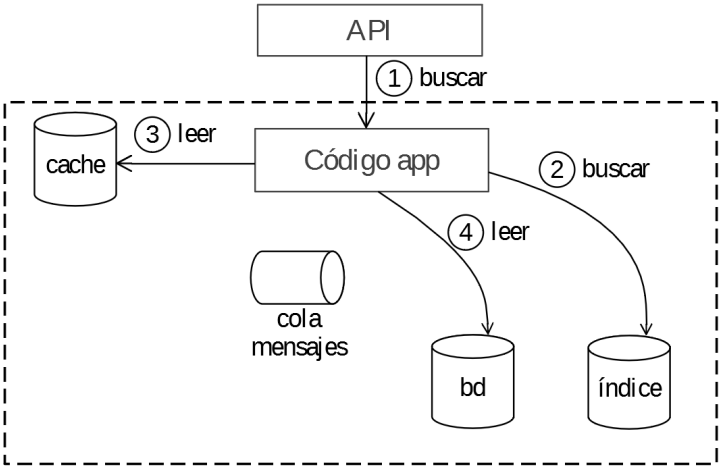
- Sistemas de tipo *Online Analytic Processing* (OLAP): efectúan operaciones de análisis de datos sobre grandes volúmenes de información. Las operaciones no suelen manipular el estado (i.e. son operaciones de sólo lectura) y es común que no se efectúen en tiempo real (i.e. no son operaciones interactivas), sino que se programan y se ejecutan en diferido (i.e. en batch). Dentro de esta categoría es posible encontrar sistemas de tipo Data Warehouse, Data Mining, o Business Intelligence.

El patrón de acceso a datos de estos dos tipos de sistemas es bien diferente. Por tanto las necesidades de los sistemas de datos son diferentes y deben diseñarse de manera distinta, optimizando diferentes cuestiones.

En el caso de los sistemas OLTP, las consultas efectuadas sobre el almacén de datos generalmente obtienen conjuntos pequeños de datos, utilizando alguna/s clave/s. También se efectúan operaciones de inserción, actualización y eliminación, basándose en los datos proporcionados por el usuario. En estos sistemas,



(a) Ejemplo de escritura



(b) Ejemplo de lectura

Figura 1.2: Ejemplo de operaciones

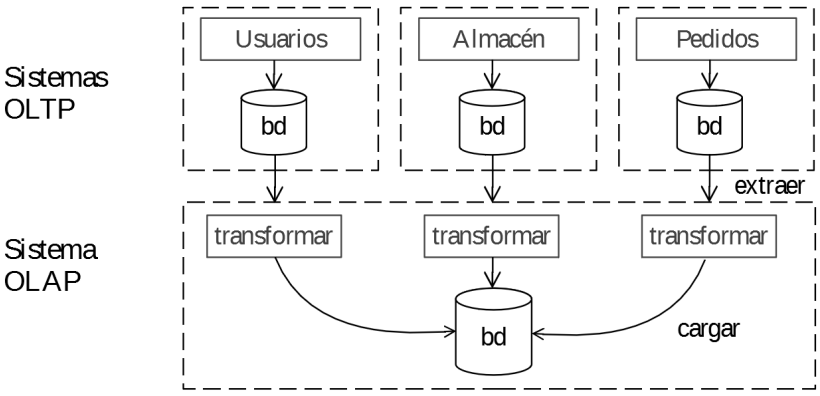


Figura 1.3: Ejemplo de sistema OLAP

la velocidad de respuesta es muy importante, por lo tanto se debe optimizar el acceso a los datos. Para ello se suelen crear estructuras adicionales denominadas *índices*, que aceleran la localización de los datos.

En el caso de los sistemas OLAP, suelen efectuarse consultas complejas, que recorren grandes volúmenes de datos y calculan valores agregados. Es habitual que estos sistemas se nutran de otros sistemas OLTP, como es el caso de los sistemas data warehouse. En estos sistemas, la información se obtiene de múltiples fuentes, por medio de procesos ETL (extract-transform-load). Aunque la velocidad de respuesta no es tan importante, para aprovechar los recursos disponibles, el sistema debe estar optimizado para trabajar con grandes volúmenes de datos. En la figura 1.3 se muestra un ejemplo en el que un sistema OLAP adquiere la información a partir de tres fuentes, constituidas por tres sistemas OLTP.

Cada sistema data-intensive posee unos requerimientos de datos muy diferentes. Es muy importante identificarlos pronto en el proceso de diseño del sistema y seleccionar las herramientas adecuadas. Para ello, resulta esencial comprender las bases de los sistemas de datos, conocer cómo organizan la información internamente, las ventajas y desventajas de cada aproximación, etc.

1.4. Propiedades

A la hora de diseñar sistemas data-intensive es recomendable maximizar tres propiedades fundamentales [Kleppmann, 2017]:

- *Fiabilidad*: el sistema debe funcionar de manera correcta incluso ante la presencia de fallos.
- *Escalabilidad*: si la carga de trabajo crece, el sistema debe ser capaz de manejar dicha carga de trabajo.

- *Mantenibilidad*: el sistema debe favorecer su mantenimiento y evolución.

En lo que sigue se analizarán con mayor detalle cada una de estas propiedades.

1.4.1. Fiabilidad

El sistema debe funcionar correctamente ante la presencia de fallos. En este contexto, *corrección* conlleva varias implicaciones:

- El sistema proporciona las funcionalidades esperadas.
- Con el rendimiento deseado.
- Lo hace de manera segura.

Respecto a los fallos, el sistema debe ser *tolerante a fallos* o *resiliente*. En general, es posible identificar los siguientes tipos de fallos:

- Fallos *hardware*: en la actualidad, cualquier centro de datos cuenta con múltiples piezas de hardware básico no especializado. La tasa de fallos de este tipo de hardware es muy habitual, y los sistemas deben ser capaces de tratar con ellos de manera habitual.
- Fallos *software*: ningún software está libre de este tipo de errores. El sistema debe efectuar auto-chequeos constantes para detectar desviaciones sobre el comportamiento esperado y el obtenido.
- Fallos *humanos*: sistemas mal configurados, errores producidos en el proceso de actualización de sistemas, etc. Los sistemas deben ser diseñados de la manera más simple posible para que resulte fácil razonar sobre ellos. Además, eliminar al máximo la interacción humana suele resultar una buena opción.

En el pasado se consideraba que el fallo era una situación anómala, y se ponían medios para tratar de evitarlos. En la actualidad, la aproximación consiste en asumir que el fallo es una situación habitual, imposible de evitar, y el sistema debe diseñarse para actuar ante él de manera natural. Por ejemplo, la compañía Netflix diseñó un componente - el *Netflix Chaos Monkey* [Izrailevsky and Tseitlin, 2011] - que es lanzado periódicamente, con el único objetivo de forzar el fallo de procesos clave de manera arbitraria. De este modo se consigue normalizar la ocurrencia de fallos en el diseño de todos sus servicios.

1.4.2. Escalabilidad

La escalabilidad tiene que ver con la habilidad del sistema de mantener su rendimiento ante cargas crecientes de trabajo, generalmente añadiendo nuevos recursos.

En este contexto, es muy importante saber determinar cuáles son los parámetros que describen la carga crítica del sistema: peticiones por segundo, tasa de lecturas o escrituras, número de sesiones concurrentes, etc.

También es importante identificar las medidas que caracterizan el rendimiento del sistema: productividad, el tiempo de respuesta, etc. Este comportamiento será generalmente modelado por medio de una distribución estadística.

Para comprobar la escalabilidad del sistema de manera empírica es habitual efectuar dos tipos de experimentos:

- Incrementar la carga del sistema manteniendo los mismos recursos, y comprobar el efecto sobre su rendimiento.
- Incrementar la carga del sistema y comprobar el nivel de recursos que se debe añadir para mantener el mismo rendimiento.

Los procedimientos habituales para incrementar los recursos de un sistema son:

- Escalado *vertical* (*scale up*): reemplazar las máquinas existentes por unas más potentes.
- Escalado *horizontal* (*scale out*): añadir nuevas máquinas y distribuir la carga entre ellas.

El escalado vertical posee límites evidentes, pero evita distribuir componentes - o datos - en distintas piezas, evitando la complejidad que ello conlleva. En la práctica se suele utilizar una combinación de ambos procedimientos.

El escalado puede efectuarse de manera manual o automática. En este último caso se habla de sistemas *elásticos*.

1.4.3. Mantenibilidad

Es conocido que la mayor parte del tiempo invertido en un sistema software es debido a su mantenimiento. Un sistema fácil de mantener contribuye a limitar al máximo la probabilidad de errores humanos.

Por lo tanto, en el diseño del sistema se deben de tener en cuenta los principales factores que afectan a la mantenibilidad de un sistema:

- *Operabilidad*: el objetivo consiste en facilitar y simplificar las operaciones que permiten al sistema funcionar día a día, por ejemplo: tareas de monitorización, trazabilidad de errores, aplicación de actualizaciones, etc.

- *Simplicidad*: si el sistema es complejo, resulta muy difícil razonar sobre él. Si el sistema resulta muy difícil de comprender, es más sencillo provocar fallos de manera accidental. La principal herramienta para luchar contra la complejidad es la abstracción. La idea consiste en ocultar las distintas partes del sistema tras ciertas abstracciones que proporcionan distintas garantías. De este modo, resulta mucho más sencillo razonar acerca del sistema.

- *Evolucionabilidad*: ningún sistema está libre de cambios. En un futuro, el sistema deberá acomodar cambios debidos a distintos factores. El diseño del sistema debe acomodar fácilmente estas actualizaciones. Este factor está íntimamente relacionado con los anteriores. Un sistema fácil de comprender será también fácil de modificar.

1.5. Procesamiento de datos

En función de la naturaleza de la información almacenada y/o procesada, un sistema de datos puede pertenecer a una de las siguientes categorías [Kleppmann, 2017]:

- Sistema de registro (*system of record*) o fuente de verdad (*source of truth*): contiene la información primaria del sistema.

- Sistemas de datos derivados: contiene información derivada de otros datos, por medio de distintos mecanismos de traducción y/o transformación.

Los sistemas de registro pueden ser encarnados por cualquier tipo de almacén (i.e. relacional, NoSQL, etc.), es decisión del arquitecto de sistemas decidir qué almacenar en qué lugar. Estos almacenes son esenciales, y su información debe ser salvaguardada a toda costa. Un fallo de estos sistemas implica la pérdida irremediable de los datos más importantes de una compañía.

Respecto a los sistemas de datos derivados, en esencia contienen información redundante, obtenida a partir de otras fuentes, y podrían ser regenerados a partir de la información primaria y/o derivada de la que proceden. En general, las técnicas de procesamiento de datos reciben una colección de *entradas* y generan una serie de *salidas*. Existen múltiples estrategias para efectuar este procesamiento. Recientemente han cobrado una relevancia extraordinaria las siguientes:

- Procesamiento en *batch*: se trata de tareas (*jobs*) que habitualmente parten de un volumen de datos muy grande y generan cierta información de salida. Son procesos que requieren bastante tiempo y son ejecutados de manera periódica. Son adecuados para tareas de análisis de datos que no requieren un resultado inmediato.

- Procesamiento en *streaming*: conforme se van produciendo eventos (o datos), estos se van procesando y se va generando información de salida. En este caso, el tiempo de procesamiento presenta una baja latencia. Pueden ser utilizados para procesar en tiempo real los datos adquiridos en alguna fuente de datos.

En las arquitecturas data-intensive modernas es habitual utilizar una combinación de ambas técnicas, obteniendo lo mejor de los dos mundos.

1.6. Arquitecturas data-intensive

Los nuevos requerimientos de datos (las 3 Vs) son muy exigentes, y las arquitecturas tradicionales no suelen resultar adecuadas. Es necesario investigar nuevas maneras de hacer. En lo que sigue se presenta primero un ejemplo que se resuelve de manera incremental con una arquitectura tradicional. Se identificarán los principales inconvenientes de esta aproximación. Después se ilustrará cómo el mismo ejemplo se puede resolver utilizando una arquitectura más adecuada, denominada *arquitectura lambda*.

1.6.1. Arquitecturas tradicionales

Para ilustrar esta situación se presentará un ejemplo en el que se desea diseñar un sistema de datos para almacenar el número de veces que cada usuario accede a una determinada URL. Se decide comenzar por un primer diseño con una base de datos relacional que consta de una única tabla con la estructura definida en la tabla 1.1. Dicha base de datos es accedida directamente por un servidor web (ver figura 1.4). Cada entrada en la base de datos refleja el número de visitas que ha recibido una determinada URL por un determinado usuario.

Si el número de operaciones de escritura crece de manera desmesurada el SGBD no será capaz de atender todas las peticiones y el sistema se saturará. Para evitar esta situación, una posible solución consiste en desacoplar el cliente del acceso a la base de datos utilizando una cola, tal y como se ilustra en la figura 1.5.

Esta solución presenta dos inconvenientes esenciales:

Nombre columna	Tipo	Descripción
id	int	Identificador de la entrada
user_id	int	Identificador del usuario
url	varchar(255)	URL accedida
count	int	Contador de visitas

Cuadro 1.1: Estructura base de datos

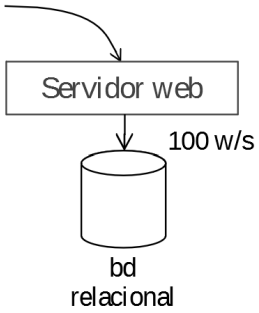


Figura 1.4: Ejemplo con SGBD

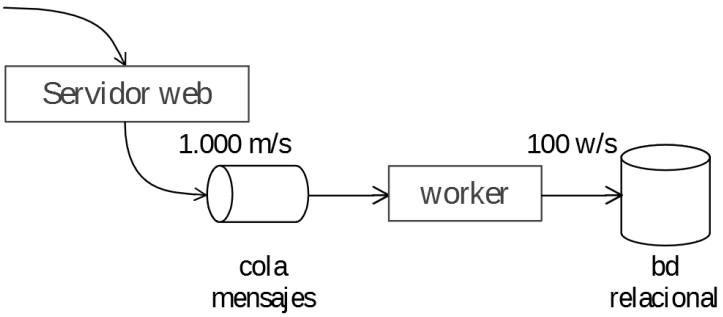


Figura 1.5: Ejemplo con cola

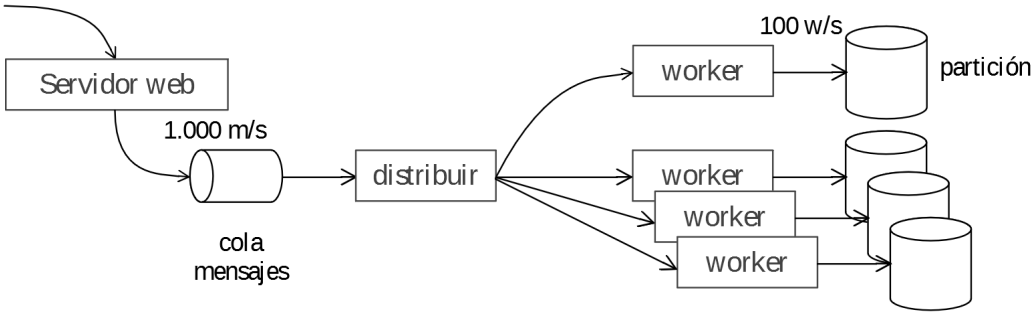


Figura 1.6: Ejemplo con particionado

- La base de datos ya no contiene información en tiempo real, pues puede existir un retardo significativo desde que se envía la escritura hasta que se escribe de manera efectiva.

- Si la carga crece más, la cola no podrá almacenar todas las operaciones de escritura y el sistema también se saturará.

Una solución consiste en *particionar* la base de datos. Este mecanismo permite dividir la base de datos en trozos independientes denominadas *particiones* o *shards*. Es necesario diseñar una estrategia de distribución de datos consistente. Además, se debe diseñar un proceso que efectúe una distribución inicial de los datos existentes, y un proceso que se encargue de determinar el nodo al que se debe enviar cada operación de escritura. De este modo se distribuye la carga entre distintos nodos (ver figura 1.6).

Sin embargo, con esta nueva arquitectura, varias cosas pueden ir mal:

- Si la carga de trabajo crece, para mantener el rendimiento es necesario reparticionar la base de datos. Para ello, las particiones se deben subdividir y se deben crear procesos que lleven a cabo toda la migración de datos, una nueva posible fuente de errores.

- Fallos humanos en la configuración de las particiones pueden causar la pérdida irremediable de datos.

- Si un nodo falla, se pierde toda la información de la partición.

Para evitar posibles pérdidas de datos o bien se efectúan copias de respaldo, backups, o bien se opta por mantener la información replicada en distintos nodos (ver figura 1.7).

Este nuevo diseño implica un sistema mucho más complejo, y ello puede dar lugar a una explosión de nuevos problemas:

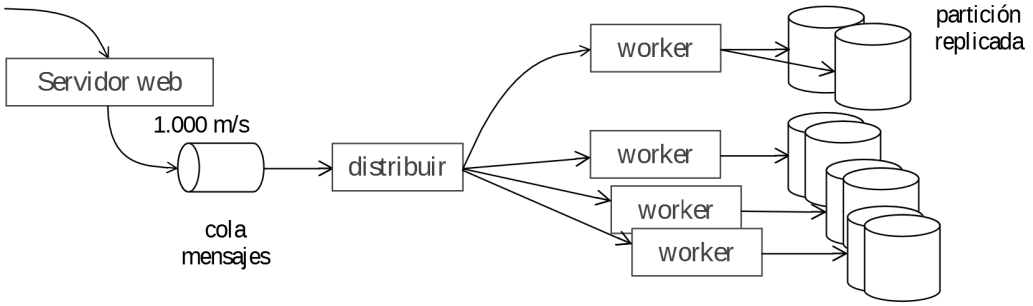


Figura 1.7: Ejemplo con replicación

- Es necesario detectar fallos en los nodos y gestionar correctamente la recuperación de los nodos que previamente fallaron.
- Retardos en la propagación de los cambios rompen la consistencia en todas los nodos, es decir, distintos nodos podrían tener distintas versiones del mismo dato.
- Fallos humanos en la configuración de los distintos nodos pueden causar la pérdida irremediable de datos.

Como se puede observar, desde la base de datos inicial, debido a las crecientes necesidades, el sistema de datos ha ido evolucionando y ganando en complejidad, hasta llegar a un punto en el que mantener los sistemas existentes ocupa la mayor parte del tiempo de los ingenieros.

1.6.2. Arquitecturas específicas

Si se prevé un gran crecimiento de carga y/o datos, es posible evitar muchos problemas utilizando algunas arquitecturas específicas desde un primer momento. Estas arquitecturas gozan de las propiedades de fiabilidad, escalabilidad y mantenibilidad perseguidas en un sistema data-intensive. Un ejemplo lo representa la *arquitectura lambda*[1].

La arquitectura lambda [Warren and Marz, 2015] parte de una (o varias) fuente de datos inmutable. La idea es que esta fuente de datos recoja todos los eventos relevantes del sistema de manera incremental, a modo de *log*. Esta fuente de datos es muy fácil de construir y posee propiedades muy deseables: las escrituras presentan muy baja latencia, la consistencia de los datos puede ser garantizada con relativa facilidad, etc.

[1] http://lambda-architecture.net/

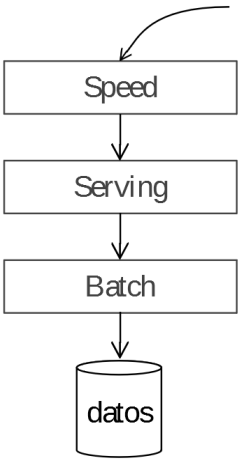


Figura 1.8: Arquitectura lambda

id	user_id	url
1	1	www.google.com
2	1	www.upv.es
3	2	www.google.com

Cuadro 1.2: Ejemplo fuente de datos

La arquitectura lambda se compone de tres capas (ver figura 1.8). Cada capa satisface una serie de propiedades y proporciona funcionalidades a su inmediata superior:

- La capa *batch*: accede a la fuente de datos que no cambia, y construye vistas derivadas con los datos requeridos por las capas superiores. Estas vistas se construyen de manera periódica, y requieren muchos recursos.
- La capa *serving*: indexa las vistas batch para poder acceder a los datos con baja latencia.
- La capa *speed*: complementa las vistas batch para obtener los datos más recientes con baja latencia.

Aplicando esta arquitectura al ejemplo del contador de visitas, cada entrada en la fuente de datos inmutable podría representar una nueva visita a una URL por parte de un usuario. En la tabla 1.2 se muestra un ejemplo de esta fuente de datos, en la que dos usuarios acceden a dos URLs diferentes.

A partir de los datos originales, en la capa *batch* se computarían los datos derivados, requeridos por las capas superiores, denominados vistas batch. En este

url	**count**
www.google.com	2
www.upv.es	1

Cuadro 1.3: Ejemplo de la vista batch visitas/url

user_id	**count**
1	2
2	1

Cuadro 1.4: Ejemplo de la vista batch visitas/usuario

caso podrían computarse algunos datos agregados, como el número de visitas por URL, o el número de visitas por usuario. Con los datos presentados en la tabla 1.2, podrían derivarse los datos listados en las tablas 1.3 y 1.4. Si la fuente de datos es muy grande, calcular estos datos supone un proceso muy costoso: se visita una a una todas las entradas de la fuente de datos y se actualiza el contador para cada URL, o para cada usuario, respectivamente. Este proceso consume muchos recursos y puede requerir bastante tiempo, por lo que se lanza periódicamente.

La capa *serving* se encargaría de indexar y publicar los datos computados por la capa batch, que reciben la denominación de vistas batch. De este modo, el acceso a estos datos derivados se produce con muy baja latencia.

El problema de esta configuración se identifica en las actualizaciones de la fuente de datos original. Como ya se ha comentado, el proceso de generación de las vistas batch es costoso y se ejecuta de manera periódica. Esto significa que el refresco de las vistas batch no es inmediato, y que las vistas pueden permanecer desactualizadas durante cierto tiempo.

Para evitar este problema se propone la capa *speed*. Esta capa recibiría en tiempo real todas las actualizaciones sobre la fuente de datos original. A partir de las vistas batch desactualizadas, aplicaría las actualizaciones recibidas en un proceso de compensación muy rápido, con el objetivo de obtener en tiempo real las vistas batch actualizadas. Estas vistas batch actualizadas por la capa speed es la que se publica a los clientes del sistema.

Eventualmente, el proceso de actualización que se ejecuta en la capa batch refrescará las vistas batch, y ya no será necesario efectuar este proceso de compensación en la capa speed. Cuando se produzcan nuevas actualizaciones, la capa speed volverá a activarse para ofrecer una versión de los datos perfectamente actualizada.

Referencias

[Izrailevsky and Tseitlin, 2011] Izrailevsky, Y. and Tseitlin, A. (2011). The netflix simian army. *The Netflix Tech Blog*.

[Kleppmann, 2017] Kleppmann, M. (2017). *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. .º'Reilly Media, Inc.".

[Warren and Marz, 2015] Warren, J. and Marz, N. (2015). *Big Data: Principles and best practices of scalable realtime data systems*. Simon and Schuster.