Liste las características de Big Data (5Vs de Big Data) Volumen: Que refiere al tamaño de los datos Velocidad: Que refiere a la velocidad con la que los datos son generados Variedad: que refiere a la amplia gama de tipos de datos Veracidad: que refiere a la confiabilidad en precision de los datos Valor: que refiere a cuanto dinero o valor se puede generar a partir de los datos	**Liste las características de Big Data (5Vs de Big Data)**
Señale las características necesarias de un sistema de Big Data Seleccione una o más de una: ☐ a. Seguridad y privacidad ☑ b. Tolerancia a fallos ☑ c. Escalable ☐ d. Interfaz amigable con el usuario ☑ e. Latencia baja de lecturas y escrituras. ☐ f. Open source	**Señale las características necesarias de un sistema de Big Data**
Señale lo correcto en lo referente a Big Data Seleccione una o más de una: ☐ a. Las fuentes de datos que intervienen en un proyecto de Big Data son por lo general fuentes homogéneas. ☑ b. Se puede considerar Big Data a un dataset que no puede ser almacenado utilizando las técnicas y herramientas tradicionales. ☑ c. Big data no solo hace referencia al almacenamiento sino también al procesamiento y análisis de grandes volúmenes de datos. ☑ d. Un dataset puede ser considerado como Big Data si supera cierto límite de tamaño, e.g., 3 Terabytes.	**Señale lo correcto en lo referente a Big Data**
Señale cuáles de los siguientes sistemas podrían ser catalogados como Big Data Seleccione una o más de una: ☑ a. Sistema para determinar el estado de salud de los cultivos mediante el análisis de imágenes tomadas por drones. ☐ b. Sistema para el manejo de inventario de la cadena Supermaxi. ☑ c. Sistema para el registro de calificaciones de los estudiantes de secundaria del Ecuador. ☑ d. Sistema de monitoreo y alerta temprana de actividad volcánica en el Ecuador.	**Señale cuáles de los siguientes sistemas podrían ser catalogados como Big Data** c, d
Señale lo correcto respecto al modelo de distribución Sharding. Seleccione una o más de una: ☐ a. Por si solo ayuda de manera notable a mejorar la capacidad de recuperación ante fallos ☐ b. Jamás se puede usar en combinación con otro modelo de distribución. ☐ c. Diferentes partes de la data se almacena en diferente servidores ☐ d. Facilita el escalamiento horizontal de lecturas y escrituras	**Señale lo correcto respecto al modelo de distribución Sharding.** c, d, b
Referente al teorema CAP, señale lo correcto. Seleccione una o más de una: ☑ a. Hablar de Consistencia sobre Disponibilidad significa garantizar la atomicidad de lecturas y escrituras rechazando algunas peticiones. ☐ b. Es posible garantizar sistemas distribuidos que sean 100% consistentes y 100%disponibles. ☑ c. Los particionamientos en la red es algo que se puede ignorar. ☑ d. La consistencia significa que ante una escritura exitosa, las lecturas posteriores siempre incluirán dicha escritura.	**Referente al teorema CAP, señale lo correcto.** a,d
Las bases de datos documentales permiten buscar únicamente por la clave Seleccione una: ◉ Verdadero ○ Falso	**Las bases de datos documentales permiten buscar únicamente por la clave** Falso

Sharding se refiere a hacer copias exactas de los datos en diferentes servidores. Seleccione una: ◉ Verdadero ○ Falso	**Sharding se refiere a hacer copias exactas de los datos en diferentes servidores.** Falso
Señales lo correcto respecto a la replicación en mongoDB Seleccione una o más de una: ☐ a. Los árbitros jamás pueden covertirse en masters si el máster actual falla. ☑ b. El máster y los esclavos pueden atender peticiones de escritura ☐ c. Los árbitros a más de almacenar una copia de los datos, votan en la elección de un nuevo master si el máster actual falla. ☑ d. En un replicaSet pueden haber esclavos que no participen en una elección.	**Señales lo correcto respecto a la replicación en mongoDB** a, c
La consistencia de sesión garantiza que durante una sesión un usuario puede leer sus propias escrituras. Seleccione una: ◉ Verdadero ○ Falso	**La consistencia de sesión garantiza que durante una sesión un usuario puede leer sus propias escrituras.**
Cuáles de los siguientes tipos de bases de datos NoSQL son orientados a la agregación Seleccione una o más de una: ☐ a. Clave-Valor ☐ b. Bases de datos orientas a columnas ☑ c. Bases de datos de Grafos ☐ d. Bases de Datos Documentales	**Cuáles de los siguientes tipos de bases de datos NoSQL son orientados a la agregación** a,b,d
Al hablar de distribución de datos se puede utilizar replicación o sharding pero nunca ambas simultáneamente. Seleccione una: ◉ Verdadero ○ Falso	**Al hablar de distribución de datos se puede utilizar replicación o sharding pero nunca ambas simultáneamente.** Falso
Enlace según corresponda Permiten ejecutar consultas basadas en la estructura interna de los agregados — BDs Clave-Valor Son útiles para almacenar datos que representan relaciones complejas como redes sociales, preferencias de usuarios, etc. — BDs de grafos No usan filas como unidad de almacenamiento y se usan en escenarios donde se necesita leer simultáneamente unas cuantas columnas de varias filas — BDs orientadas a columnas Colección de objetos relacionados que queremos tratar como unidad de manipulación de datos y consistencia — Agregación Generalmente no permiten ejecutar consultas basadas en los campos de una agregación — BDs de grafos	**Enlace según corresponda** ● Documental ● Grafo ● Columna ● Agregacion ● *
El teorema CAP hace referencia a ciertas características que deben cumplir los sistemas distribuidos. Señale cuáles son dichas características Seleccione una o más de una: ☐ a. Rendimiento ☑ b. Tolerancia a particionamientos en la red ☐ c. Tolerancia a fallos ☑ d. Consistencia ☐ e. Usabilidad ☑ f. Disponibilidad	**El teorema CAP hace referencia a ciertas características que deben cumplir los sistemas distribuidos. Señale cuáles son dichas características**

Resumen Big Data
Capítulo 1

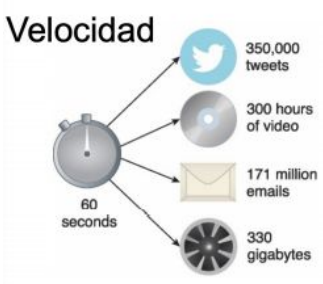
¿Qué es Big Data?

- *Big Data se dedica al análisis, procesamiento y almacenamiento de una gran cantidad de datos provenientes de fuentes heterogéneas.*
- Big data Consiste en conjuntos de datos que crecen tanto que resulta difícil trabajar con ellos utilizando herramientas de administración de bases de datos disponibles (Wikipedia).
- Big data es cuando el tamaño de los datos en sí se convierte en parte del problema. (Mike Lukides, O'Reilly Radar)
- No son solo sus problemas de "Big Data", se trata de sus GRANDES problemas de "datos" (Alexander Stojanovic, Hadoop Manager on Win Azure)

Campo de aplicación de Big Data

- Optimización de operaciones
- Identificación de nuevos mercados
- Predicción (Clima, desastres, bolsa de valores)
- Detección de fraudes
- Soporte a la toma de decisiones
- Descubrimientos científicos

Las 5 vs de BIG DATA

- Volumen: El tamaño de la data
- Velocidad: La velocidad con la que la data se genera
    - 

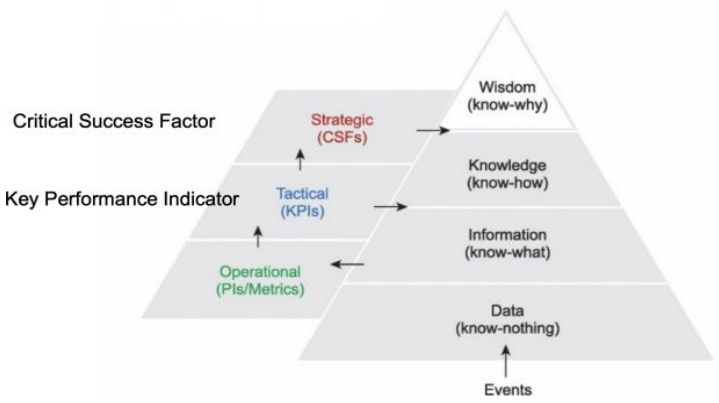
- Variedad: Los diferentes tipos de data
    - 

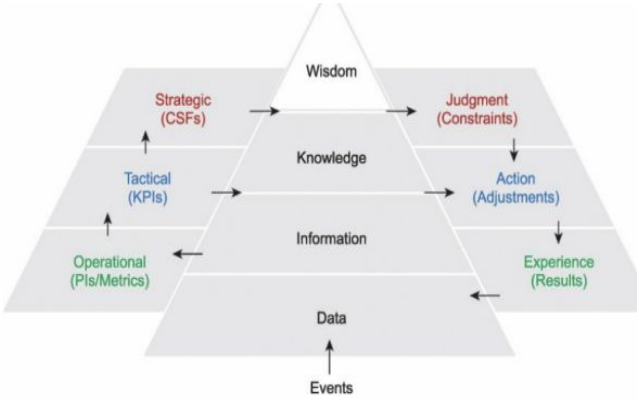
- Veracidad: La confiabilidad de los datos en términos de precisión
    - Twitter
- Valor: Big Data es útil cuando se puede convertir en valor (Lucrar de los datos)

Causas para la adopción de Big Data

- ***Nueva dinámica del mercado***
    - Burbuja .com, recesión 2008.
    - Mantener rentabilidad y Reducir costos
    - Mantener nuevos y antiguos clientes, mediante nuevo productos/servicios o valor agregado al cliente

- ***Arquitectura del negocio***





- ***Manejo de procesos de negocios***
    - Descripción de cómo se realiza el trabajo.
    - Actividades del negocio y las relaciones con los actores responsables de ejecutarlas.
    - Procesos alineados a los objetivos del negocio
- ***TICs***
    - Análisis de datos y ciencia de datos
    - Digitalización
    - Tecnología asequible y hardware básico
    - Medios de comunicación social
    - Comunidades y dispositivos hiperconectados/ Hyper-connected communities and devices
        - Se relaciona directo con la Internet de las cosas (IoT)
        - Permitir obtener información de todos los dispositivos posibles.
        - Utilizar como fuente de datos comunidades de información.
    - Computación en la nube
        - Capacidad de utilizar la nube para acceder a los datos
        - Utilizar la capacidad de procesamiento existente en aplicaciones almacenadas en la nube.
- ***Internet of Everything (IoE)***
    - 14 billones
    - 2020: 32 billones

Características deseadas de un sistema de Big Data

- ***Robustez y tolerancia a fallos***
    - *El sistema se comporta correctamente aunque algunos PCs se han caído.*

    - Compleja semántica y consistencia en base de datos distribuidas.
    - Los sistemas deben ser ”human-fault-tolerant”-tolerante a fallo humano
- ***Latencia baja en lecturas y escrituras***
    - *Leer/Escribir mucha información en muy pocos segundos.*
    - Algunas aplicaciones requieren tiempo para propagar las actualizaciones en sus sistemas.
    - Se requiere leer rápidamente información sin comprometer la robustez del sistema
- ***Escalabilidad***
    - *Capacidad de agregar nuevos datos o recursos sin comprometer el desempeño del sistema.*
    - La arquitectura Lambda es horizontalmente escalable a través de cada capa.
    - Se logra al añadir varias computadoras.
- ***Generalizable***
    - *Puede soportar un número grande de aplicaciones.*
    - La arquitectura Lambda esta basada en función de todos los datos.
    - Los datos pueden ser de diferente tipo: financieros, social media, aplicaciones cientıficas.
- ***Extendible***
    - *No reinventar la rueda cada vez que se quiera agregar una característica.*
    - *Agregar una funcionalidad requiere un costo mínimo de esfuerzo.*
    - A veces la inclusión de una nueva funcionalidad requiere de la migración de datos viejos en un nuevo formato.
    - *Capaz de migrar grandes cantidades de datos rápida y fácilmente.*
- ***Ad hoc queries***
    - La posibilidad de *crear consultas específicas* para obtener información interesante.
- ***Mantenimiento Mínimo***
- ***Depurable***

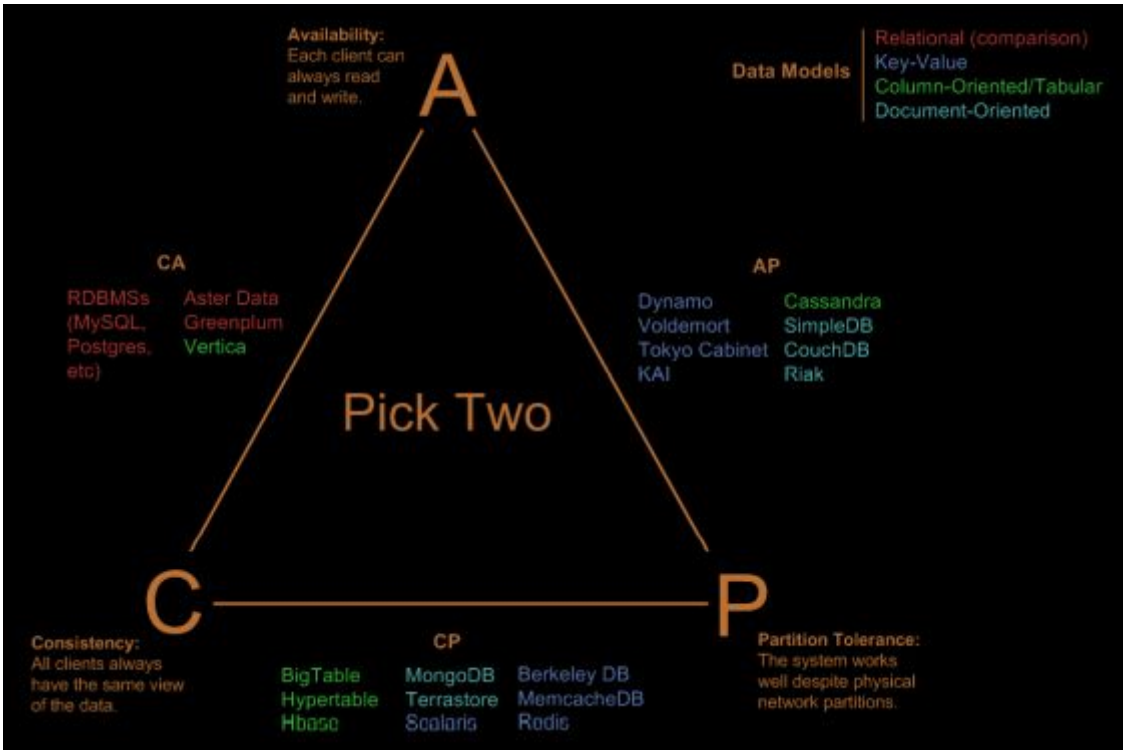
Soluciones de Big Data

- ***MapReduce:*** Framework de hardware distribuido (cluster/grids) que divide los problemas en subproblemas (Map) y luego se recopila las mini-respuestas (Reduce) para generar conclusiones. La solución más común es Hadoop. Modelo creado y promovido por Google.
- ***NoSQL Database***
    - *Amplia clase de sistemas de gestión de bases de datos* que difieren del modelo clásico del sistema de gestión de bases de datos relacionales (RDBMS) en múltiples aspectos:
        - No usan SQL como principal lenguaje de consultas
        - Los datos almacenados *no requieren estructuras fijas como tablas*
        - *No garantizan ACID* (atomicidad, consistencia, aislamiento y durabilidad)
        - *Escalan bien horizontalmente* (ej: MongoDB, Cassandra, BigTable)
- ***Algoritmos genéticos***
    - *Un algoritmo es una serie de pasos organizados que describen el proceso que se debe seguir, para dar solución a un problema específico.*
    - Con la inteligencia artificial, surgieron los algoritmos genéticos, inspirados en la evolución biológica
    - Evolucionan sometidos a mutaciones y recombinaciones genéticas.
- ***Recaptcha (Google)***
    - reCAPTCHA es una extensión de la prueba CAPTCHA.

    - Reconocer texto presente en imágenes, usado para determinar si el usuario es o no humano.
    - Mejorar la digitalización de textos.
- ***Reconocimiento de patrones***
    - *Ciencia que se ocupa de los procesos sobre:*
        - Ingenieria, computacion y matematicas
        - Relacionados con objetos fisicos o abstractos
        - Extracción de información que permita establecer propiedades de entre conjuntos de dichos objetos.
- ***NUI (Natural User Interface)***
    - *Interfaz que permite interactuar con un sistema sin utilizar sistemas de mando o dispositivos de entrada de las GUI, usando en su lugar movimientos gestuales*.
    - Ej: Kinect. Reconocimiento de gestos y movimientos.

Teorema CAP

- Consistency (Consistencia), Availability (Disponibilidad) y Partition Tolerance (Tolerancia al Particionamiento)
- Nos dice que en un sistema distribuido de almacenamiento de datos no podemos garantizar consistencia y disponibilidad (para actualizaciones)
- Partición (queda separado en dos o más islas).
- Depende de las exigencias del proyecto para saber que atributos de calidad es necesario y elegible.



El teorema es que solo puedes garantizar dos de estos tres atributos:

CP (Consistencia y Tolerancia al particionamiento):

- No disponibilidad
- No es elegible con clientes que requieren que el sistema esté disponible 100% del tiempo o muy cerca

- Se puede lograr en cierto nivel, pero el **sistema esta enfocado en aplicar los cambios de forma consistente aunque se pierda comunicación con algunos nodos**.

AP (Disponibilidad y Tolerancia al particionamiento):

- No garantiza datos iguales en todos los nodos todo el tiempo
- En este caso el **sistema siempre estará disponible para las peticiones aunque se pierda la comunicación entre los nodos**.

CA (Consistencia y disponibilidad):

- No particionado de los datos, porque se **garantiza que los datos siempre son iguales y el sistema estará disponible respondiendo todas las peticiones**.
- Por ejemplo, los sistemas de bases de datos relacionales (SQL) son CA porque todas las escrituras y lecturas se hacen sobre la misma copia de los datos.

Presentaciones

- ***Cassandra:*** Es clave valor, CAP es AP
- ***CouchDB:*** Es documental, CAP es AP
- ***Mongo:*** Es documental, CAP depende contexto, mayoritariamente CP
- ***Firebase:*** Es documental organizado en colecciones, CAP es CP
- ***REDIS:*** Es clave valor, Redis es CP xq deja de estar disponible en particiones minoritarias

Algoritmos genéticos

Los algoritmos genéticos se inspiran en la evolución natural para solucionar problemas de optimización que de otra forma serían difíciles para un diseñador humano.Se llama población al conjunto de soluciones e individuo a cada una de las soluciones. El algoritmo evalúa cada una de las soluciones y selecciona las que mejor resuelvan el problema.

Un algoritmo genético es cuando se usan mecanismos que simulan los de la evolución de las especies de la biología para formular dichos pasos. Es una técnica de inteligencia artificial inspirada en la idea de que el que sobrevive es el que está mejor adaptado al medio, es decir la misma que subyace a la teoría de la evolución que formuló Charles Darwin.

Cualquier algoritmo genético puede ser resumido en 6 pasos:

- Inicialización
- Evaluación
- Selección
- Reproducción
- Crossover
- Mutación

Uso de los algoritmos genéticos en Big Data para:

- Clasificar los datos de manera eficiente.
- Extraer la información relevante de la gran cantidad de datos como paso previo para luego aplicar el algoritmo de clasificación.
- Es más robusto que los modelos de inteligencia artificial

Bases de datos NoSQL

Cassandra

Cassandra se define como una base de datos NoSQL distribuida que permite manejar grandes volúmenes de datos. Su objetivo principal es la escalabilidad lineal y la disponibilidad. Combina propiedades de una base de datos clave-valor y una orientada a columnas. Podemos interactuar con Cassandra mediante CQL a través de la shell. de CQL, cqlshell.

Características:

- **Protección de datos sólida:** un diseño de registro de confirmación evita la pérdida de datos.
- **Consistencia de los datos sintonizables**
- **Replicación de datos multi-centro:** se trata de un centro de datos transversal (en diferentes zonas geográficas)
- **Compresión de datos:** garantiza que los datos se comprimirán hasta un 80%
- **CQL (Lenguaje de Consulta Cassandra):** un lenguaje similar a SQL

Redis

Redis es un motor de base de datos en memoria, basado en el almacenamiento en tablas de hashes (clave/valor) pero que opcionalmente puede ser usada como una base de datos durable o persistente.

Ventajas:

- Una velocidad muy alta, gracias a su almacenamiento en memoria
- Posibilidad de persistir datos en disco para recuperación ante fallas
- Fácil configuración
- Alta disponibilidad
- Curva de aprendizaje baja
- Una variedad de tipos de datos

Desventajas:

- El método de persistencia RDB consume mucho I/O (escritura en disco)
- Todos los datos trabajados deben encajar en la memoria (en caso de no usar persistencia física)

Firebase

FireStore es una base de datos NoSQL de ***tipo documentos*** de Firebase. Cloud Firestore es una base de datos NoSQL alojada en la nube, flexible y escalable para la programación en servidores, dispositivos móviles y la web desde Firebase y Google Cloud Plataform. Estos documentos se organizan en ***colecciones***, que son contenedores para los ***documentos*** y se pueden usar para organizar los datos y compilar consultas.

Esta base de datos NoSQL usa ***CP***(Consistencia y Tolerancia al particionamiento) del teorema CAP.

CouchDB

Gestor de base de datos NoSQL de código abierto. Desarrollado por Apache Software Foundation en 2005. Compatible con los sistemas operativos Linux, Unix, MacOS y Windows. Escrito en el lenguaje de programación Erlang. Guarda los datos en forma de documentos.

Esta base de datos NoSQL usa AP(Disponibilidad y Tolerancia al particionamiento) del teorema CAP, además tiene una consistencia parcial mediante replicación y verificación.

MongoDB

MongoDB es un sistema de base de datos NoSQL orientado a documentos de código abierto y escrito en C++. Al ser de código abierto es multiplataforma. MongoDB solo garantiza operaciones atómicas a nivel de documento, además aquí no se usan los joins, si se requiere consultar dos o más colecciones(tablas) se debe realizar más de una consulta. Si se necesita realizar consultas de agregación MongoDB tiene un Framework para realizar consultas llamado Aggregation Framework.

MongoDB es una base de datos orientada a documentos, es decir que en lugar de guardar los datos en registros, guarda los datos en documentos.

Características:

- Consultas ad hoc
- Indexación
- Replicación
- Balanceo de carga
- Almacenamiento de archivos
- Ejecución de JavaScript del lado del servidor

MongoDB en el teorema CAP, no se puede simplemente decir que MongoDB es CP / AP / CA, porque en realidad es una compensación entre C, A y P, dependiendo de la configuración de la base de datos / controlador y el tipo de criterio.

Escenario	Enfoque principal	Descripción
No particion	CA	El sistema está disponible y proporciona una gran consistencia
mayoría conectada / Partición	AP	Las escrituras no sincronizadas del antiguo primario se ignoran
Partición / Mayoría no conectada	CP	solo se proporciona acceso de lectura para evitar sistemas separados e inconsistentes

Resumen Cap 2

Data Model

- El modelo con el cual la base de datos organiza su información Modelo de Datos Relacional
    - Conjunto de tablas
        - Filas que representan alguna entidad
        - Columnas: describen a la entidad (cada columna un solo valor).
        - Cada columna puede apuntar a otra fila en la misma o en otra tabla, lo que representa una relación entre esas dos entidades

Modelos de Datos NoSQL

Se aleja del Modelo Relacional

Cada solución NoSQL tiene su propio modelo

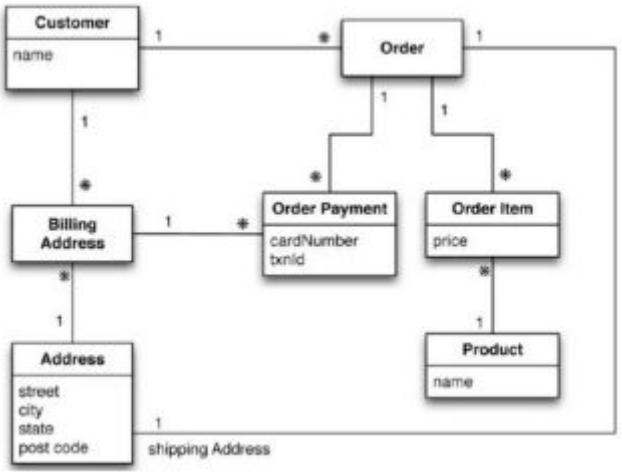
Existen 4 categorías

1. Clave - Valor = Orientadas a la Agregación
2. Documentales = Orientadas a la Agregación
3. Orientadas a columnas = Orientadas a la Agregación
4. Grafos

Agregados

- Modelo Relacional
    - Tuplas (Estructura de datos limitada)
        - No hay como anidar una tupla dentro de otra
        - Peor aún poner una lista de valores o tuplas dentro de otra
        - Esta simplicidad permite realizar todos los tipos de operaciones conocidas sobre las tuplas.
- Normalmente se necesita trabajar con datos en unidades que tienen una estructura más compleja que un conjunto de tuplas.
- Se puede pensar en un registro complejo que permita anidar listas y otras estructuras dentro de él.
- DBs Clave-Valor, Documentales y Orientadas a Columnas usan esta clase de “registros complejos”
- Un agregado es una colección de datos relacionados que nosotros queremos tratar como una unidad.
- **Unidad** para la manipulación de datos y para el manejo de la consistencia.
- Actualizar agregados con operaciones atómicas y comunicarse con la base de datos en términos de agregados.
- Facilitan la ejecución en un cluster de computadores, ya que un agregado sería la unidad para replicación y “sharding” o fragmentación
- Facilitan el trabajo a los desarrolladores

Ejemplo: Relaciones y Agregados



Customer

Id	Name
1	Martin

Orders

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

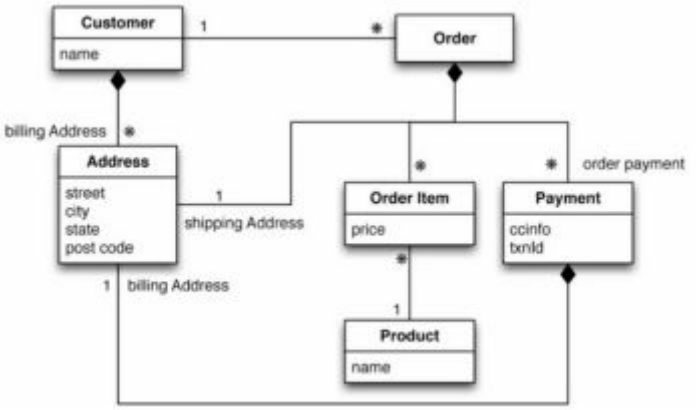
Id	OrderId	ProductId	Price
100	99	27	32.45

Address

Id	City
77	Chicago

OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abeli*879rft



```
// in customers
{
   "id":1,
   "name":"Martin",
   "billingAddress":[{"city":"Chicago"}]
}
```

```
// in orders
{
   "id":99,
   "customerId":1,
   "orderItems":[
      {
         "productId":27,
         "price": 32.45,
         "productName": "NoSQL    Distilled"
      }
   ],
   "shippingAddress":[{"city":"Chicago"}]
   "orderPayment":[
      {
         "ccinfo":"1000-1000-1000-1000",
         "txnId":"abelif879rft",
         "billingAddress": {"city": "Chicago"}
      }
   ],
}
```

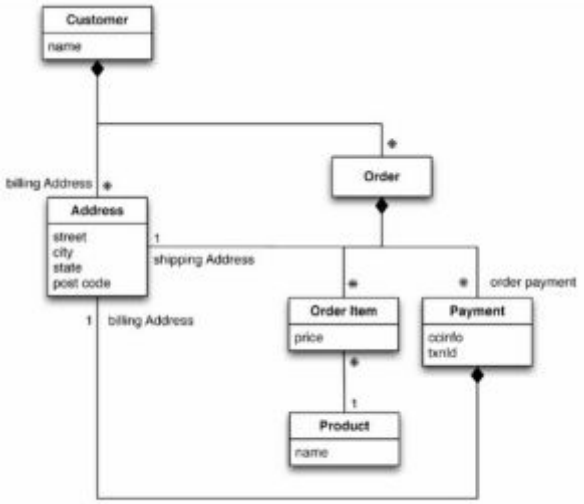


Figure 2.4. Embed all the objects for customer and the customer's orders

```
// in customers
{
   "customer": {
      "id": 1,
      "name": "Martin",
      "billingAddress": [{"city": "Chicago"}],
      "orders": [
      {
         "id":99,
         "orderItems":[
         {
            "productId":27,
            "price": 32.45,
            "productName": "NoSQL Distilled"
         }],
         "shippingAddress":[{"city":"Chicago"}]
         "orderPayment":[
         {
            "ccinfo":"1000-1000-1000-1000",
            "txnId":"abelif879rft",
            "billingAddress": {"city": "Chicago"}
         }],
      }]
   }
}
```

Consecuencias de la Agregación

- El modelo relacional captura los datos y sus relaciones muy bien, pero sin ninguna noción de una unidad de agregación.
- Las técnicas propuestas por las tecnologías NoSQL nos permiten crear agregaciones o estructuras compuestas. Pero....
    - Los que modelan no proporcionan ninguna información para distinguir una agregación de otra
    - Si la hay, esta varıa dependiendo de cada situación

- Cuando se trabaja con BDs orientadas a agregación, lo único claro es que la agregación es la unidad de interacción con la base de datos.
- Pero todo depende de cómo los datos son usados dentro de la aplicación. (Esto a veces se sale de las manos de la modelación)
- RDBMS y de Grafos (Aggregate-ignorant) → (No conocen la agregación)
    - NO es el fin del mundo
- Es difícil definir los límites de las agregaciones correctamente, especialmente si la misma data es usada en diferentes contextos
    - Por ejemplo: Agregación Orden o Producto?
- La agregación facilita la ejecución en clusters.
- Al definir agregaciones le decimos a la base de datos que bits van a ser manipulados juntos
- Y por lo tanto deberían residir en el mismo nodo.
- RDBMS → ACID (Tablas, Filas)
- NoSQL → BASE (Una agregación a la vez)
    - Varias Agregaciones de manera atómica

DBs Clave-Valor y Documentales

- *Ambas son fuertemente orientadas a agregación*
- Ambas bases consisten de varias agregaciones donde cada agregación tiene una clave o ID que es usada para recuperar los datos
- Diferencia Clave-Valor:
    - La agregación es desconocida para la base de datos
    - Blob (Binary Large Objects) de bits sin ningún significado
    - Se puede almacenar lo que sea. (Límite en espacio)
- Documentales:
    - Es capaz de distinguir la estructura dentro de las agregaciones.
    - Ciertos límites en lo que se puede almacenar
        - Estructuras y tipos predefinidos
        - Flexibilidad en el acceso
- Diferencia
    - Clave-Valor: Permite buscar agregaciones por Clave o ID
    - Documentales: Queries basados en la estructura interna de la agregación. Puede ser la clave, pero por lo general es otro valor.

BDs Orientadas a Columnas

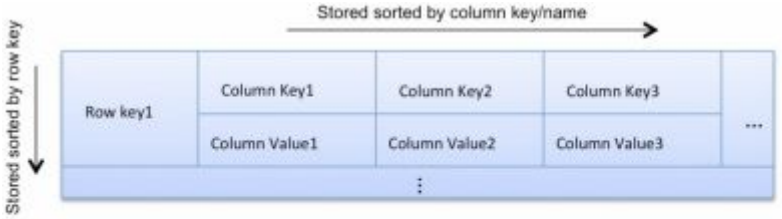
- RDBMS
    - Filas como unidad de almacenamiento
        - Ayuda a mejorar el rendimiento en las escrituras
    - Hay muchos escenarios donde las escrituras son esporádicas:
        - Se necesita leer algunas columnas de varias filas a la vez.
        - Grupos de columnas para todas las filas como unidad de almacenamiento.

De-Moneda	A-Moneda	Precio venta	Precio Compra	Banco	Fecha
USD	EURO	1.2300	1.2850	Austro	18/09/2014
USD	PLN	1.3400	1.3050	Austro	18/09/2014
USD	AZN	1.2700	1.5150	Austro	18/09/2014
USD	EGP	1.7600	1.5800	Austro	18/09/2014
USD	EEK	1.4000	1.4213	Austro	18/09/2014
USD	FJD	1.4425	1.4553	Machala	18/09/2014
USD	GIP	1.4681	1.4929	Machala	19/09/2014
USD	DKK	1.5177	1.4874	Machala	19/09/2014
EURO	USD	1.4571	1.4642	Machala	19/09/2014
EURO	FJD	1.4713	1.4749	Pichincha	19/09/2014
EURO	GIP	1.4785	1.4799	Pichincha	19/09/2014
EURO	EEK	1.4812	1.4766	Pichincha	19/09/2014

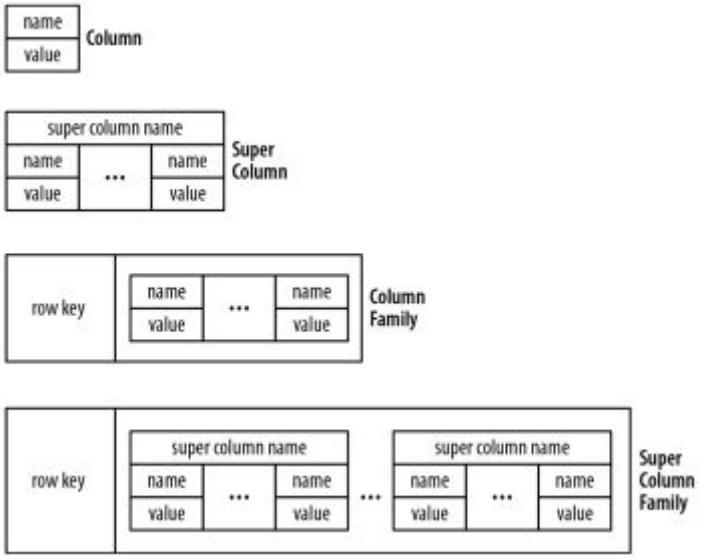
- 1B, 100Bytes = 100GB x (3/6); 100MB/sec = 500sec
    - Almacenamiento más eficiente debido a la compresión de datos USDx8, EUROx4 Funciones de Agregación Max, Min, Sum, Avg

De-Moneda
USD
USD
USD
USD
USD
USD
USD
USD
EURO
EURO
EURO
EURO

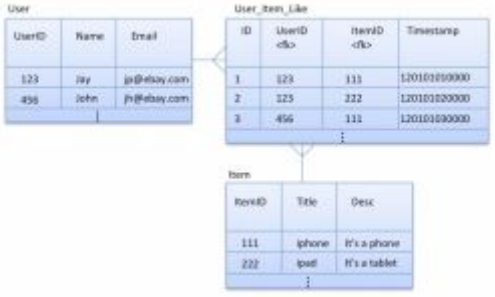
- Estructura de agregación a dos niveles



Map<RowKey, **SortedMap**<ColumnKey, ColumnValue>>



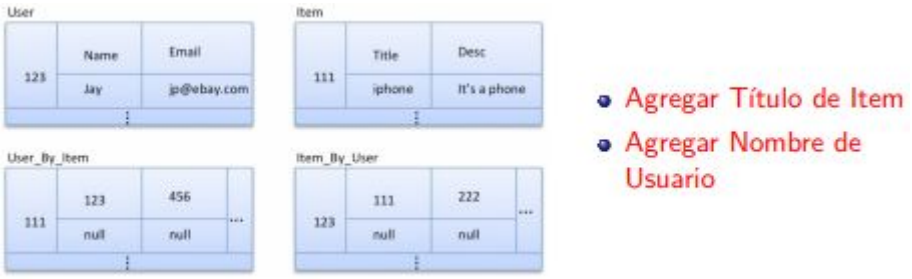
- Modelar de acuerdo a los patrones de búsqueda
    - Pensar en patrones de búsqueda por adelantado
        - Identificar los queries más frecuentes
        - Detectar cuales pueden ser lentos
        - Asegurarse que el modelo satisface los queries más frecuentes y crıticos
    - Recordar que una “Familia de Columnas” es un SortedMap
        - Búsqueda, Ordenamiento, Agrupamiento, Filtrado, Agregaciones, etc.
- De-Normalizar y Duplicar por lecturas eficientes
    - No de normalizar si no se necesita, encontrar un balance
- Normalización
    - Pros: No hay duplicación de información, menos errores por modificación de datos, conceptualmente mas claro, etc...
    - Cons: Queries demasiado lentos si hay varios joins y demasiados datos (Big Data)
- Likes: Relación entre usuarios e ítems



- Usuarios x UserID
- Items x ID
- Todos los items que le gustan a un usuario en particular
- Todos los usuarios a los que les gusta un ítem en particular
- Réplica del modelo relacional

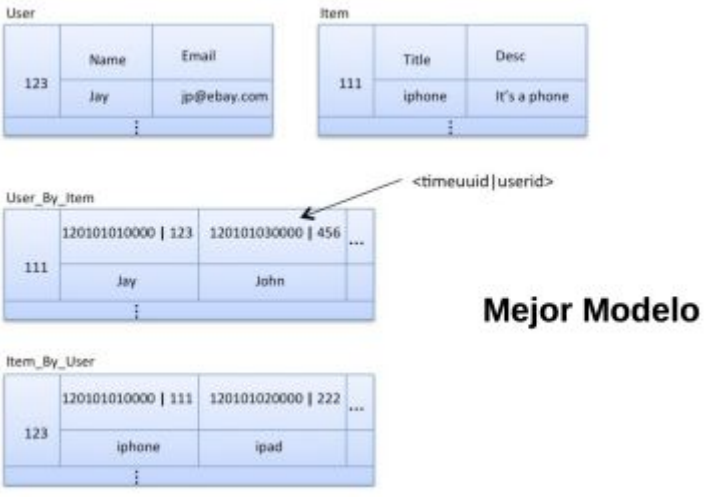
- Usuarios x UserID
- Items x ID

- Entidades Normalizadas con ındices personalizados



- Usuarios x UserID
- Items x ID
- Todos los items que le gustan a un usuario en particular
- Todos los usuarios a los que les gusta un ítem en particular

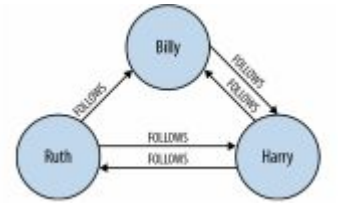
- Entidades Normalizadas con de-normalización en ındices personalizados



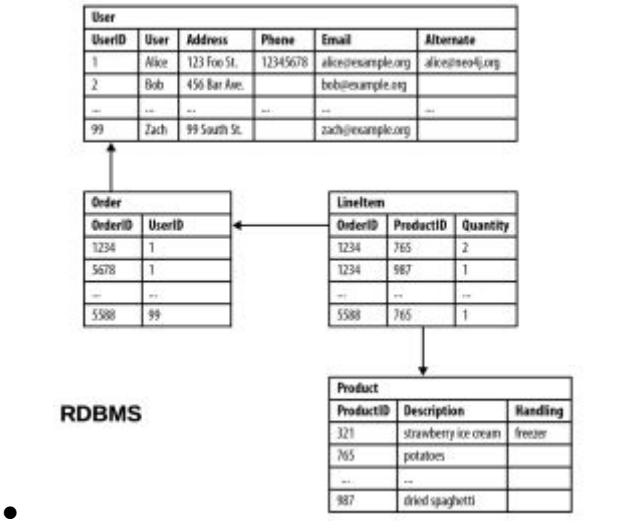
- **Resumen**

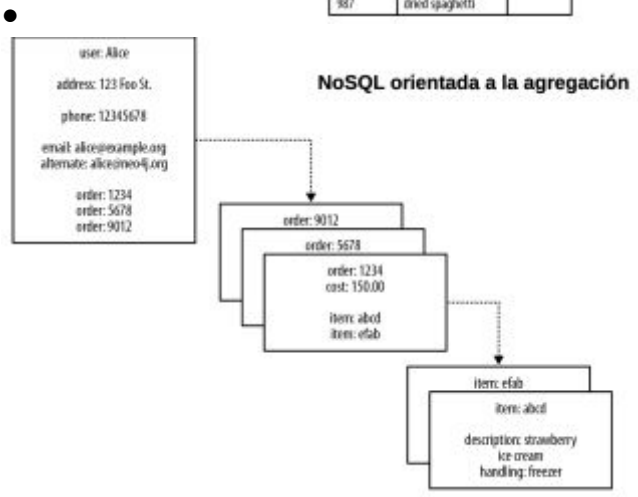
    - Un agregado es una colección de datos relacionados que queremos tratar como unidad de almacenamiento
    - BDs Clave-Valor, Documentales, Orientadas a Columnas son orientadas a la agregación
    - Agregados permiten la ejecución en clusters de servidores
    - Las BDs orientadas a agregación funcionan muy bien cuando las interacciones con los datos son hechas con la misma agregación.
    - Aggregate-Ignorant BDs son mejores cuando las interacciones con la base usan datos que están organizados de múltiples formas.

Bases de datos de Grafos

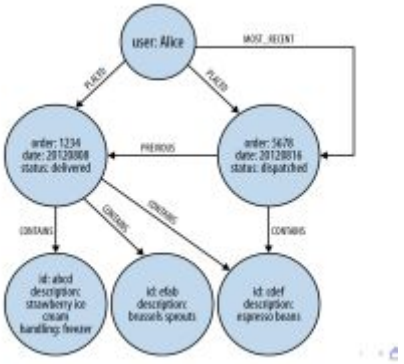
- Cómo manejar pequẽnos registros que tienen conexiones complejas entre śı?
- Modelo de datos de Grafos
    - Nodos y Relaciones Nodos tienen propiedades
    - Las relaciones son nombradas y directas y siempre tienen principio y fin
    - Las relaciones también pueden tener propiedades
- 

- Base de datos cuya estructura de almacenamiento sigue el modelo de datos de grafos.
- Sirven para capturar datos que consisten de relaciones bastante complejas:
    - Redes sociales
    - Preferencias de productos
    - Control de Tráfico
    - Detección de fraude
    - Etc...
- Ejemplos:
    - Deme todos los restaurantes que han sido recomendados por mis amigos
    - Cual es el camino más corto para ir de Cuenca a Machala?
    - Cuales son los productos que a John y Ana les gustan.
- Por que son útiles
    - RDBMS y NoSQL orientadas a agregación no permite capturar relaciones complejas en sus modelos de datos.





- Recomendaciones
  - Quien compra helado de fresa también compra Cafe
- Enriquecer nuestros datos
  - Unirlo a grafos de otros dominios (Geo y Social)
- Todos los sabores de helado que le gusta a la gente que vive cerca de Juan y a las que les gusta el espresso pero no les gusta las coles de bruselas



Bases de datos sin esquema

- RDBMS
  - Se debe definir un esquema de antemano
  - Una estructura definida que nos diga que tablas y columnas existen y que tipo de datos va a tener cada columna.

- NoSQL (el almacenamiento es más casual)
    - Clave-Valor: permite almacenar cualquier tipo de datos
    - Documentales: No hay restricciones en la estructura de los documentos
    - Columnas: permiten almacenar cualquier tipo de datos en cualquier columna que se desee.
    - Grafos: permite agregar libremente nodos y aristas y propiedades a los nodos y las aristas.
- Con un esquema se tiene que definir por adelantado lo que se necesita almacenar.
- Sin un esquema que nos ate, se puede almacenar fácilmente lo que sea que se necesite.
- Esto permite cambiar fácilmente nuestra base de datos a medida que se conoce más el proyecto.
- Se puede fácilmente agregar nuevas cosas a medida que se las va descubriendo
- Si no se necesita algo, simplemente se deja de almacenarlo.
- Así como permitir cambios, una BD sin esquema permite manejar fácilmente datos no-uniformes.
    - Datos donde cada registro tiene diferentes campos.
- Un esquema pone a todas las filas en una “camisa de fuerza”.
- Que pasa si hay diferentes tipos de datos en cada fila?
    - Se termina con columnas en valor null
    - O columnas sin sentido (Columna 4)
- NoSQL evita esto permitiendo que cada registro tenga lo que necesita.
- Ni más ni menos
- Las BDs sin esquema pueden evitarnos muchos problemas que existen con las bases que tienen un esquema fijo
- Sin embargo,estas traen algunos problemas consigo.
    - Los programas necesitan saber que la dirección de facturación es llamada direccionDeFacturacion y no direccionParaFacturacion
    - Y que la cantidad es un entero 5 y no cinco
- El hecho es que cuanto escribimos un programa que accede a ciertos datos, este siempre depende implícitamente de un esquema.
- Un programa asume
    - Que ciertos nombres de campos se encuentran presentes y que estos hacen a referencia a datos con cierto significado
    - Algo acerca del tipo de datos almacenado en ese campo.
- Los programas no son humanos
    - No deducir que Nro = Numero, a menos que se los programe para hacer eso.
- Así tengamos una BD sin esquema, siempre está presente el esquema implícito.
- **Esquema Implıcito** es un conjunto de suposiciones acerca de la estructura de los datos en el código que manipula esos datos.
- ***Esquema Implıcito presenta algunos problemas.***
    - Para entender que datos se están manejando hay que escarbar dentro del código.
    - Si el código es bien estructurado se puede definir fácilmente el esquema, pero nadie nos garantiza eso
    - La BD no puede usar el esquema para recuperar o almacenar los datos más eficientemente.
    - La BD no puede validar los datos para evitar inconsistencias.
- Las BDs sin esquema mueven el esquema al código de la aplicación

    - Que pasa si diferentes aplicaciones hechas por diferentes personas acceden a la misma base de datos?
- Para reducir los problemas
    - Encapsular la interacción con la BD dentro de una sola aplicación que se integra con otras usando servicios web.
    - Delimitar diferentes partes de un agregado para las diferentes aplicaciones
        - Diferentes secciones de un documento
        - Diferentes "Familia de columnas"
- Usar una base de datos sin esquema no es la panacea
- No uniformidad en los datos es una buena razón para usar una BD sin esquema

Vistas materializadas

- Ventajas de los modelos orientados a agregación
    - Si se quiere acceder a órdenes, es conveniente tener todos los datos de una orden en un agregado que será almacenado y leído como una unidad
- ¿Hay desventajas?
    - Que pasa si queremos saber cuánto se ha vendido un producto en las últimas semanas?
    - La agregación juega en contra. Hay que obligatoriamente leer cada orden para responder a esa pregunta
- RDBMS tienen la ventaja que permiten acceder a los datos de diferente manera
- Además proporcionan un mecanismo que permite observar los datos de manera distinta a la que están almacenados: Vistas
- Una vista es como una tabla pero es calculada a partir de las tablas base
- Cuando se accede a la vista la BD calcula los datos a mostrarse en la vista
- Hay algunas vistas que son costosas de calcular
- Solucion: Vistas Materializadas (VM)
    - Son vistas precalculadas y son almacenadas en caché en disco.
    - Son efectivas para datos que son de lectura-intensa pero que pueden estar algo desactualizados.
- NoSQL no tiene vistas materializadas pero pueden tener queries precalculados en una caché en disco.
- Las VM son un aspecto central en la BDs orientas a la agregación, tal vez mas que las RDBMS
    - Las aplicaciones tienen que ejecutar queries que no se adaptan al modelo de agregación
    - Es usual crear vistas materializadas usando **Map-Reduce**
- Dos estrategias para crear vistas materializadas
    - Actualizar la VM al mismo tiempo que se actualiza los datos base.
    - Ejecutar procesos en batch que actualizan la VM en intervalos regulares
- Se puede calcular externamente, leyendo los datos calculando la vista y grabando de nuevo en la base de datos
- Es importante conocer el modelo de negocios para saber cuan desactualizada puede estar una VM

Modelos de Distribución

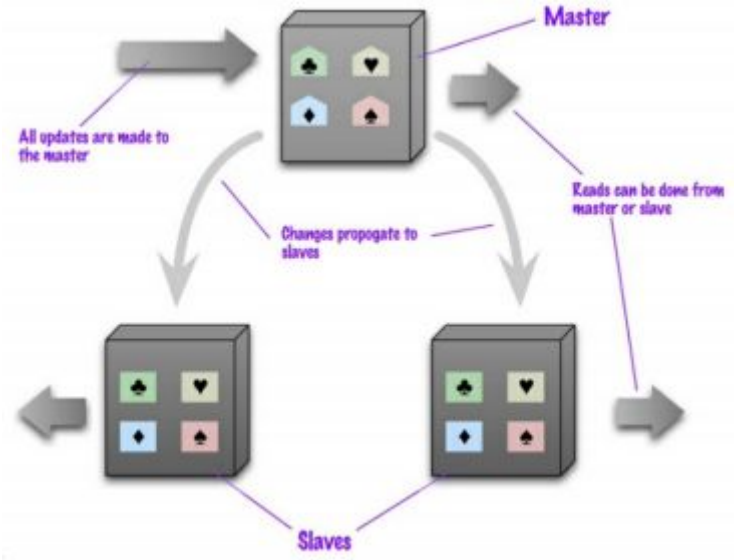
- NoSQL DBs se ejecutan en un cluster de servidores.
    - + volumen de datos
    - + complejidad en escalar verticalmente

    - NoSQL permite escalar horizontalmente.
    - La agregación es la unidad de distribución
- Dependiendo del modelo de distribución:
    - Manejar volúmenes de datos mas grandes
    - Procesar mayor numero de reads o writes
    - Mayor disponibilidad en caso de retardos en la red o particionamiento de red.
- Estos beneficios traen un costo (complejidad) Se debe usar un cluster cuando los beneficios son evidentes.
- Replicación
    - Hace copias exactas de los datos en diferentes servidores
- Sharding
    - Diferente data en diferentes nodos
- Se puede usar cada una por separado o los dos a la vez

Modelos de Distribución

Único Servidor , Maestro-Esclavo (Replicación), Sharding, Peer-to-Peer (Replicación)

- **Unico Servidor**
    - Sin distribución de datos
    - Ejecutar la BD en una sola máquina que maneja todos las lecturas y escrituras
    - Si bien las BDs NoSQL fueron diseñadas con la idea de ejecutarse en clusters también se lo puede hacer en un único servidor (BDs Grafos)
    - Si la interacción con los datos es en su mayoría mediante el uso de agregaciones:
        - BDs Único Servidor Documentales o Clave-Valor
    - No hay que dejarse sorprender por la palabra Big Data.
    - Si se puede manejar los datos sin distribución, optar siempre por la opción mas simple:
        - Único Servidor.
- **Maestro-Esclavo (Replicación)**



    - Mas útil para datasets de lectura intensa y pocas escrituras
    - Garantiza las lecturas (read resilience)
        - Si el master falla, los esclavos pueden seguir manejando peticiones de lectura (si la mayoría de accesos son de lectura)

- Si falla el master no hay escrituras
    - Hasta que el master se recupere Se designa un nuevo máster
- Facilidad de reemplazar al master por un esclavo (incluso sin necesidad de escalar)
- Es como un único-servidor con un respaldo en caliente (hot backup).
    - Único servidor con mayor tolerancia a fallos
    - Para garantizar lecturas, se debe asegurar que los paths de lecturas y escrituras sean diferentes
- El master puede ser seleccionado manual o automáticamente (Leader Election).
- Replicación tiene ventajas pero tiene su inevitable lado negativo: inconsistencia
    - Diferentes clientes leyendo diferentes esclavos pueden ver datos diferentes
    - Algunos clientes no ven los ultimos cambios del sistema
    - Si falla el master, todos los datos no sincronizados se pierden

- **Sharding**

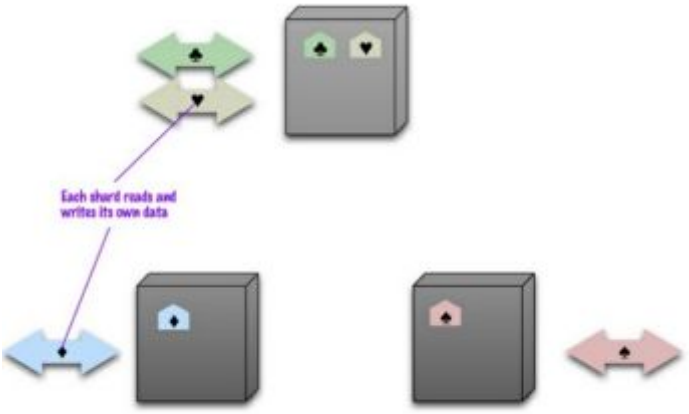


Figure 4.1. Sharding puts different data on separate nodes, each of which does its own reads and writes.

- Normalmente una BD esta ocupada porque diferentes usuarios están accediendo a diferentes partes de los datos
- Se puede soportar escalamiento horizontal poniendo diferentes partes de la data en diferente servidores (Sharding)
- En el caso ideal cada usuario se comunica con un solo servidor, obteniendo respuestas rápidas de ese servidor. (10 servidores, 10% c/u)
- Para lograr algo cercano al caso ideal
    - Los datos que son accedidos simultáneamente sean almacenados juntos en el mismo servidor
        - Agregaciones: unidad de distribución
- Factores para mejorar el rendimiento
    - Si el acceso esta basado en ubicación física, se puede poner los datos lo mas cerca posible al lugar desde donde son accedidos.
    - Tratar de balancear la carga entre los servidores
    - Poner diferentes agregados juntos si se sabe que se van a leer en secuencia
- Sharding puede ser manual (lógica de la aplicación) o automática.
- Permite escalar horizontalmente lectura y escrituras
    - Sharding no ayuda a mejorar la capacidad de recuperación ante fallos
    - Ante fallos solo los usuarios que acceden al shard sufrirán las consecuencias
- Pero no es bueno perder parte de los datos
- Sharding es mas fácil de lograr con las agregaciones, pero no se debe tomar a la ligera

- Algunas bases son dise˜nadas para usar sharding, en este caso usar sharding desde el principio
- Otras bases no fueron dise˜nadas para eso, pero permiten pasar de único-servidor a sharding
- Saber identificar cuando se requiere usar sharding y usarlo mucho antes de que realmente se necesita

- **Peer-to-Peer (Replicacion)**

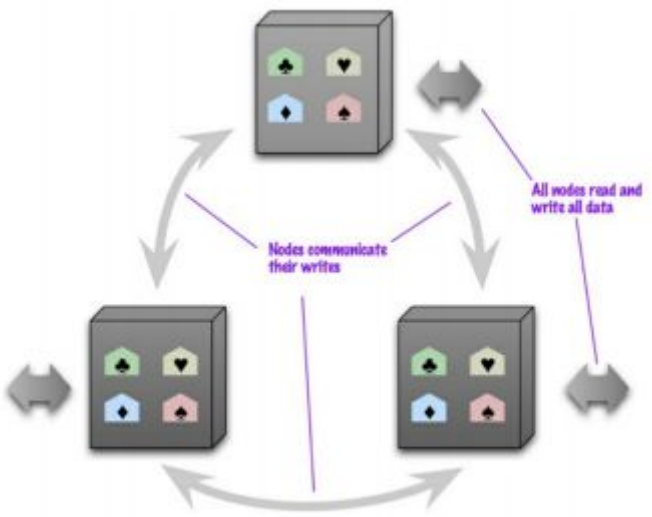


Figure 4.3. Peer-to-peer replication has all nodes applying reads and writes to all the data.

- **Combinar Sharding y Replicación**

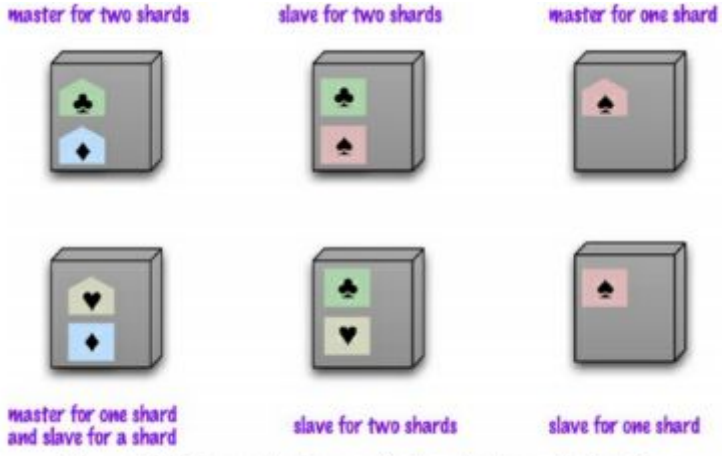


Figure 4.4. Using master-slave replication together with sharding

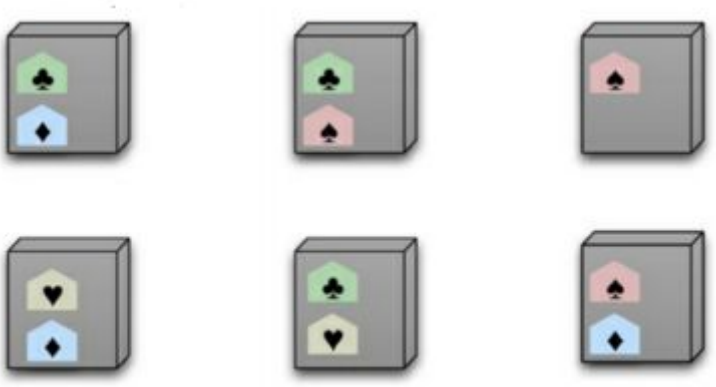
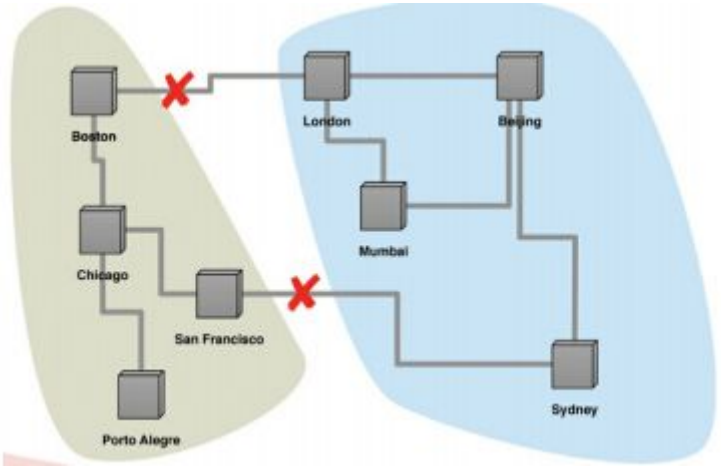


Figure 4.5. Using peer-to-peer replication together with sharding

-
- **Peer to peer**
    - Todas la réplicas tienen el mismo peso, todas aceptan escrituras y la pérdida de una de ellas no afecta la lectura de los datos
    - Complicación: Consistencia
        - Dos clientes tratando de escribir el mismo registro al mismo tiempo (conflicto de escritura)
        - Solución: Coordinar las escrituras.
            - No se necesita que todas se pongan de acuerdo pero si la mayoría

Teorema CAP

- Dr. Eric Brewer, 2000: Teorema CAP (1)
- Un sistema distribuido con datos compartidos puede tener a lo mucho dos de las siguientes propiedades:
    - Consistencia (Consistency)
    - Disponibilidad (Availability)
    - Tolerancia a Particionamientos de red (Partition tolerance).
- Gilbert and Lynch, 2002: Una definición mas formal y pruebas (2)
    - (1) Harvest, Yield, and Scalable Tolerant Systems
    - (2) Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web
- **Consistencia**: Después de una escritura exitosa, las lecturas posteriores siempre incluyen dicha escritura.
- **Disponibilidad**: Siempre se puede leer y escribir en el sistema. Toda petición (lectura/escritura) recibida por un nodo que se encuentra operativo debe proporcionar una respuesta.
- **Tolerancia a Particionamientos de Red**: La red podrá perder arbitrariamente varios mensajes enviados de un nodo a otro en presencia de particiones (Gilbert and Lynch)

- 

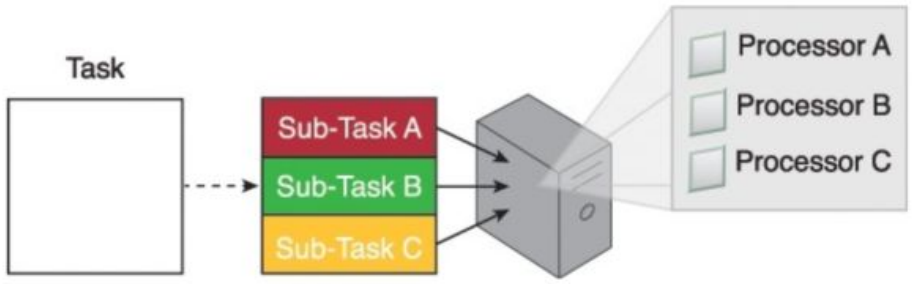
- ***Particionamiento: No solo paquetes perdidos***
    - Servidor caido (Una partición)
        - Todos los paquetes enviados hacia el se pierden.
        - La falla mas sencilla de manejar (Hay seguridad que el servidor no da respuestas erróneas)
- ***CA Systems***
    - Consistencia, Disponibilidad – No Particionamiento
        - Único Servidor: Sistema Monolıtico (No red, no particion y CA garantizadas)
        - ¿Clientes conectados a ese servidor?
    - Sistemas Distribuidos que sea CA (Multi Nodo)
        - Los mensajes en la red nunca se pierden o retrasan
        - Los servidores nunca se caen
    - Sistemas ası NO EXISTEN
    - Ante la presencia particiones de red (Errores), que se sacrificara? Consistencia o Disponibilidad?
        - La posibilidad de particiones siempre esta presente
    - La decisión no es mutuamente exclusiva
        - El sistema no será completamente consistente ni completamente disponible
        - Combinación de las dos que se adapta a las necesidades.
- ***Consistencia sobre Disponibilidad***
    - Garantiza la atomicidad de lecturas y escrituras rechazando algunas peticiones.
        - Bajar el sistema por completo
        - Rechazar escrituras (Two-Phase Commit)
        - Lecturas y escrituras en las particiones cuyo master esta en esa partición.
- ***Disponibilidad sobre Consistencia***
    - Responderá a todas las peticiones
        - Lecturas desactualizadas
        - Aceptando conflictos de escritura → Mecanismos para resolver inconsistencias (vector clocks)
        - Hay varios sistemas donde es posible manejar resolución conflictos y en los cuales lecturas desactualizadas son aceptables.
- ***Disponibilidad o Consistencia nunca ambas***
    - Sistema que dice ser CA : Servidores A, B y C
        - {A,B} {C}

        - Petición de escritura a C para actualizar un dato
        - 1. Aceptar la escritura sabiendo que ni A ni B sabrán de ella (hasta que la red vuelva a la normalidad)
        - 2.Rechazar la escritura, sabiendo que el cliente no podrá contactar A o B (hasta que la red vuelva a la normalidad)
    - Se debe escoger Disponibilidad (opcion 1) o Consistencia (opción 2)
    - En lugar de pensar cual de las dos propiedades nuestro sistema requiere (CA), pensar hasta donde puedo sacrificar cada una, antes que mi sistema empiece a funcionar mal.
    - No importa lo que hagamos siempre habrá fallas en el sistema.
        - Dejar de responder peticiones (lectura/escritura)
        - Dar respuestas basadas en información incompleta

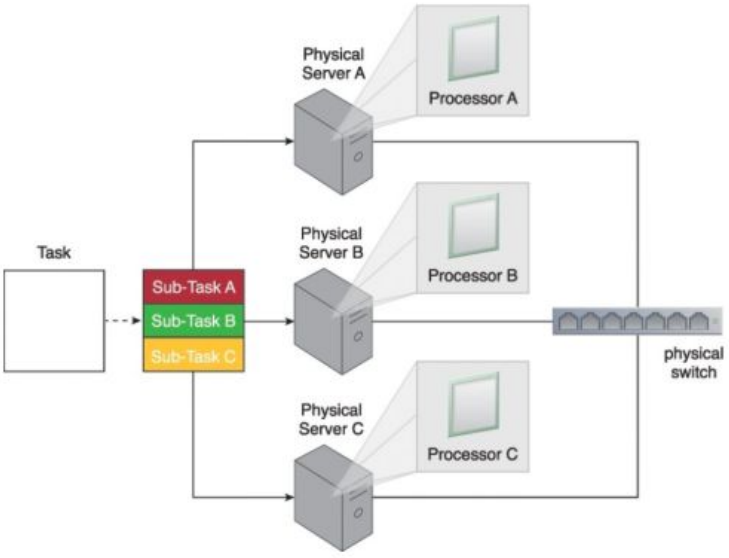
Capítulo 3

Conceptos y Técnicas de Procesamiento de Big Data

Procesamiento Paralelo



Procesamiento Distribuido



Actualización síncrona

- Con actualizaciones sincrónicas, los clientes se comunican directamente con la base de datos y bloquean hasta que se completa la actualización

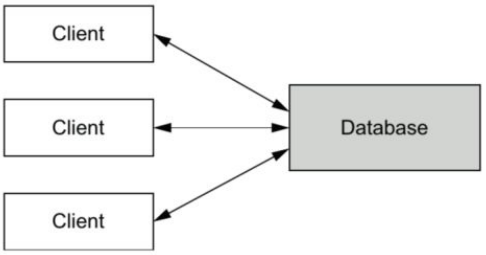


Fig: Arquitectura de capa de velocidad simple que utiliza actualizaciones sincrónicas

- Con actualizaciones síncronas, los clientes envían las actualizaciones a la cola e inmediatamente proceden con otras tareas.
- Después de un tiempo, el stream processor lee un lote de mensajes de la cola y aplica las actualizaciones

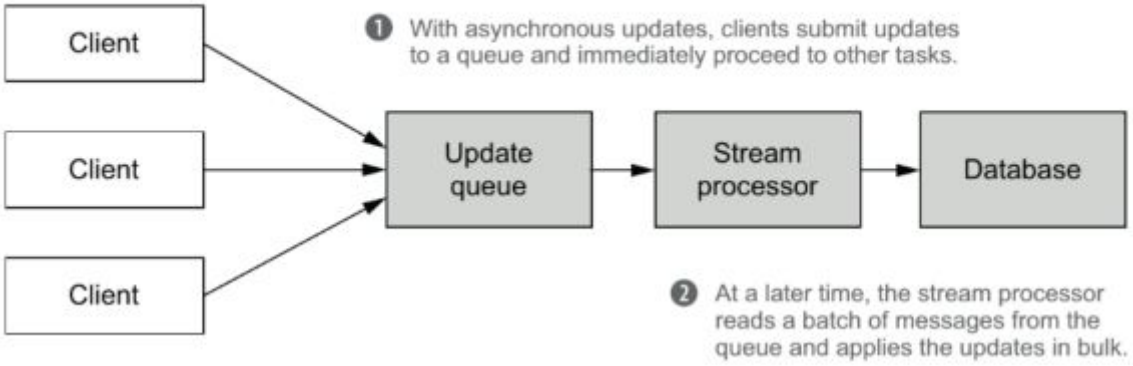


Fig x: Actualizaciones asincrónicas proporcionan mayor rendimiento y manejan fácilmente cargas variables

Colas y Procesamiento de Streams

- Arquitecturas Asíncronas
    - Colas
    - Stream Processing
- Procesamiento sin colas persistentes
    - Dispara y Olvida (Fire and Forget)
    - Tráfico?

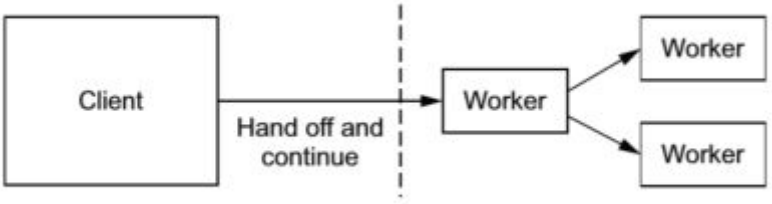


Fig: Para implementar procesamiento asincrónico sin colas, un cliente envía un evento sin monitorear si su procesamiento es exitoso.

Servidor de colas de único consumidor - Single-consumer queue server

- Los mensajes se eliminan de la cola cuando son confirmados

- Múltiples aplicaciones consumiendo los mismos eventos?
- El problema es que la cola controla que fue consumido y que no

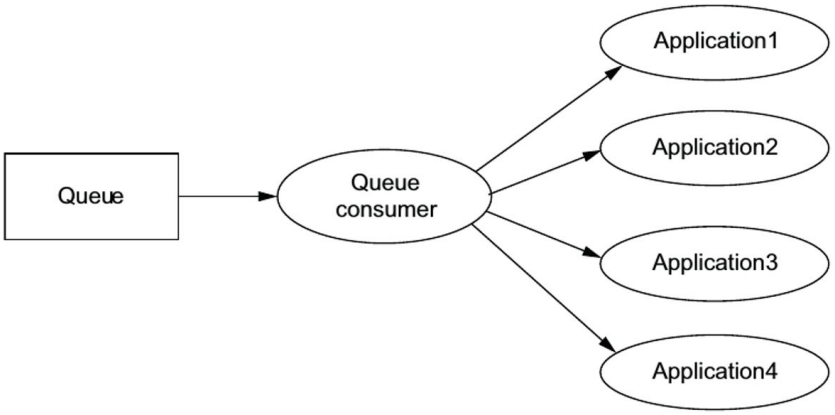


Figure 14.2 Multiple applications sharing a single queue consumer

Multi-consumer queues

Pasar el control de los eventos consumidos a la aplicación. Con cola multiconsumidor, las aplicaciones requieren ítems específicos de la cola y es responsable del tracking de los procesos exitosos de cada evento

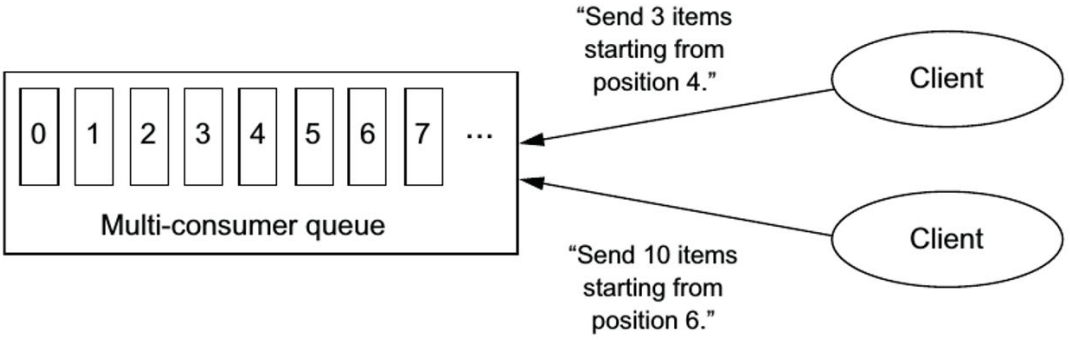


Figure 14.3 With a multi-consumer queue, applications request specific items from the queue and are responsible for tracking the successful processing of each event.

Stream processing

- Procesar los eventos y actualizar las vistas en tiempo real
    - Uno a la vez
    - Micro Batches

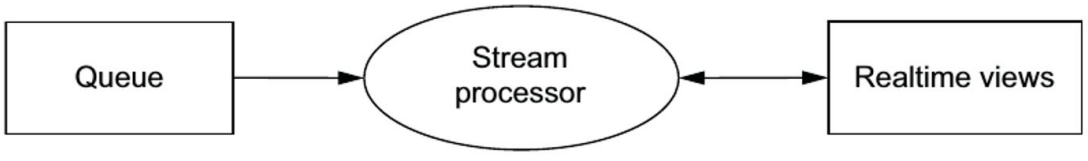


Figure 14.4 Stream processing

	One-at-a-time Uno a la vez	Micro-batched Micro-por lotes
Baja latencia	✔	
Alto Rendimiento		✔
Semántica al menos una vez	✔	✔
Semántica exactamente una vez	en algunos casos	✔
Modelo de programación simple	✔	

Colas y Procesadores(Workers)

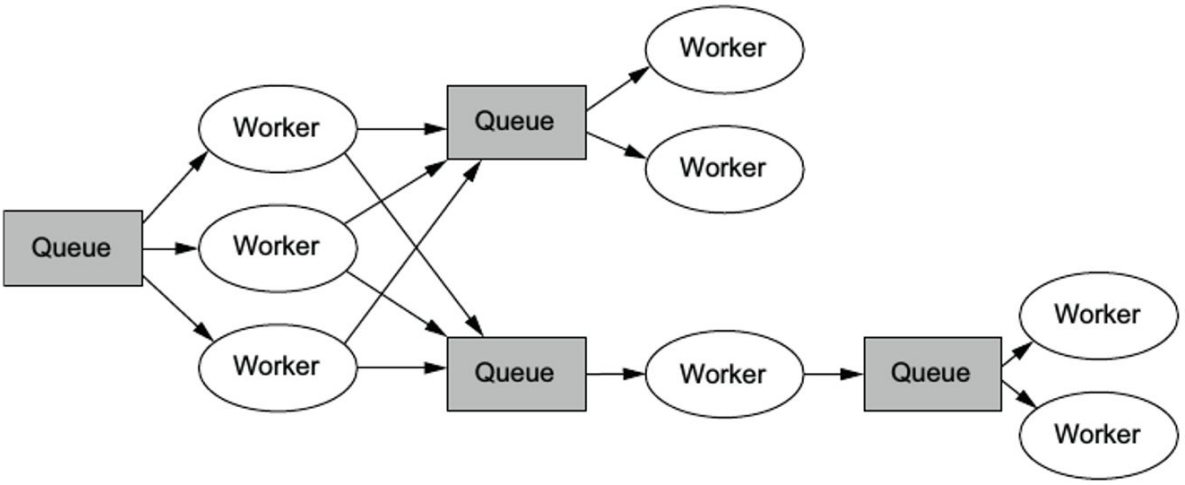


Figure 14.6 A representative system using a queues-and-workers architecture. The queues in the diagram could potentially be distributed queues as well.

Un sistema representativo usado en arquitectura de colas y trabajadores. Las colas en el diagrama pueden potencialmente ser colas distribuidas

Capıtulo 3 - Map Reduce

Big problems

- Facebook con 10 mil millones de fotos (x4: 40 mil millones de archivos), un petabyte en total; Se agregan 2-3 terabytes cada día (2008)
- Bolsa de Nueva York: un terabyte de datos comerciales por día
- El Archivo de Internet: se agregan 100 terabytes por mes; 3 petabytes en total (2009)
- El Gran Colisionador de Hadrones en Ginebra, Suiza produce ~ 15 petabytes por año
- La Web: 100 mil millones de páginas web → 400-500 terabytes comprimidos (duplicados en varios clústeres)
- eBay tiene 6,5 PB de datos de usuario + 50 TB / día (2009)

Parallel Algorithm

Un algoritmo que

- se puede ejecutar una pieza a la vez
- en muchos dispositivos de procesamiento diferentes
- luego volver a armar
- para obtener el resultado correcto

Un problema que demora 4 meses en una máquina, solo tomará 3 horas en 1000 máquinas

Challenges

- Identificar el trabajo que se puede realizar al mismo tiempo
- Sin dependencia de datos entre subproblemas
- Mapeo del trabajo a las unidades de procesamiento
- Distribuyendo el trabajo
- Administrar el acceso a los datos compartidos
- Sincronizar varias etapas de ejecución
- Recopilación de resultados Tolerancia a fallos

Programming Efforts

Trabajo por hacer:

- Comunicación y coordinación
- Recuperarse de un fallo de la máquina
- Informe de estado
- Depuración
- Mejoramiento
- Localidad

Esto debe repetirse para cada problema distribuido que desee resolver.

Prelude to MapReduce

MapReduce es un paradigma diseñado por Google para hacer que un subconjunto (grande) de problemas distribuidos sea más fácil de codificar

Automatiza la distribución de datos y la agregación de resultados

Restringe las formas en que los datos pueden interactuar para eliminar bloqueos.

Proporciona una interfaz genérica que oculta la distribución y la complejidad de los cálculos a gran escala.

Map and Reduce

Dos funciones principales:

- Map: toma datos y crea registros de datos interesantes
- Reduce: toma datos interesantes del mapeador y los resume

El esquema permanece igual, mapee y reduzca el cambio para adaptarse al problema

History

Los conceptos de MAP y REDUCE provienen de la programación funcional (Lisp, ML - Metalanguage)

Framework lanzado internamente en Google en 2003.

En 2004, Jeffrey Dean y Sanjay Ghemawat publicaron un artículo "MapReduce: procesamiento de datos simplificado en grandes clústeres"

En 2007, primera implementación de código abierto (Hadoop)

En 2010, el primer taller internacional sobre MapReduce y sus aplicaciones (MAPREDUCE'10)

MapReduce Provides

Distribución y paralelización automática

- Reduce la complejidad de la sincronización
- Divide datos automáticamente
- Maneja el equilibrio de carga
- Optimización de la transferencia de red y disco

Tolerancia a fallos

- Proporciona transparencia de fallas

Herramientas de estado y monitoreo

Abstracción limpia para programadores

Elimina muchas preocupaciones de confiabilidad de la lógica de aplicación

Parallel Computing at Google

Otros sistemas de cosecha propia

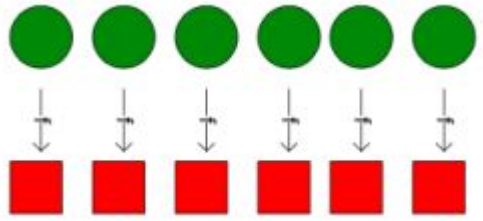
- Google File System (GFS): un sistema de archivos distribuido tolerante a fallas
- BigTable: una base de datos distribuida tolerante a fallos

Alternativa de Hadoop:

- HDFS
- HBase

Functional Programming - Map

map function applies a function f1 to each value of a sequence



map input:

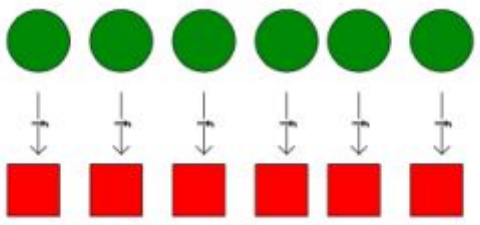
Unary function with parameter of type T1 that produces output of type T2

Array of elements of type T1

map output:

Array of the same size of elements of type T2

map function applies a function f1 to each value of a sequence



Lisp:

```
(define (square n)
      (* n n))

(map square '(1 2 3 4 5))
```

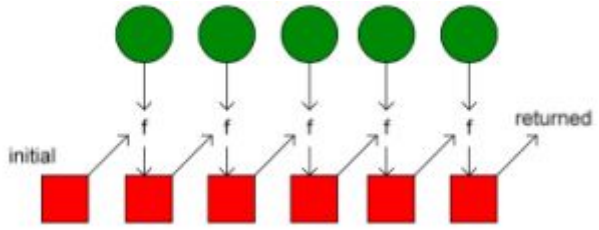
Scala:

```
def square(n:Int):Int = n*n

(1 to 5).map(x => square(x))
```

```
Output:(1 2 3 4 5) => (1 4 9 16 25)
```

Functional Programming - Reduce

reduce combines all elements of a sequence using a binary operator f2



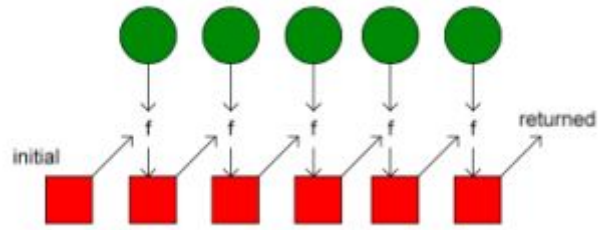
reduce input:

Binary function with parameters of type (T1, T2) that produces output of type T2
Initial element of type T2
Array of elements of type T1

reduce output:

Element of type T2

reduce combines all elements of a sequence using a binary operator f2



Lisp:

```
(define (plus a b)
      (+ a b))

(reduce plus 0 '(1 2 3 4 5))
```

Scala:

```
def plus(n1:Int, n2:Int):Int
                = n1+n2

(1 to 5).foldLeft(0)(
        (x,y) => plus(x,y))
```

```
Output:(1 2 3 4 5) => (((((0+1)+2)+3)+4)+5) => 15
```

Data Flow in MapReduce

Leer datos de entrada

Map:

- Extraiga algo que le interese de cada registro
- Particione la salida: qué teclas van a qué reductor

Shuffle and Sort: el reductor espera que sus claves estén ordenadas y para cada clave: lista de todos los valores

Reducir:

- Aggregate, summarize, filter, or transform

Escribe los resultados

Key / Value Pairs

```
map (in_key, in_value) ->
  list (out_key, intermediate_value)
```

Processes input key/value pair

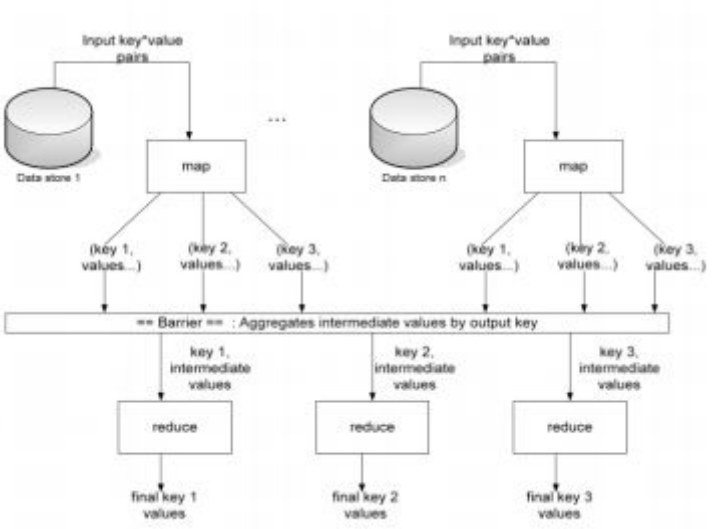
Produces set of intermediate pairs

```
reduce (out_key,list<intermediate_value>)
  -> (out_key, out_value)
```

Combina todos los valores intermedios para una clave en particular
Produce un valor de salida combinado (también puede ser una lista)

Map
Los registros de la fuente de datos (líneas de archivos, filas de una base de datos, etc.) se introducen en la función de mapa como key*value pairs: e.g., (filename, line).
Map produce uno o más valores intermedios junto con una clave de salida de la entrada.

Reduce
Una vez finalizada la fase de mapa, todos los valores intermedios para una clave de salida determinada se combinan en una lista.
reduce combina esos valores intermedios en uno o más valores finales para esa misma clave de salida (en la práctica, generalmente solo un valor final por clave)



Parallelism
funciones map se ejecutan en paralelo, creando diferentes valores intermedios a partir de diferentes conjuntos de datos de entrada
funciones reduce también se ejecutan en paralelo, cada una trabajando en una tecla de salida diferente
Todos los valores se procesan de forma independiente
Bottleneck: la fase de reducción no puede comenzar hasta que la fase del mapa haya finalizado por completo.

Example: Count Word Occurrences

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values)
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
 Emit(AsString(result));
```

```
O my Love's like a red, red rose
That's newly sprung in June;
O my Love's like the melodie
That's sweetly played in tune.
                (R. Burns, 1794)

MAP 1:
<a, 1>
<like, 1>
<Love, 1>
<my, 1>
<O, 1>
<red, 1>
<red, 1>
<rose, 1>
<'s, 1>

MAP 2:
<in, 1>
<June, 1>
<newly, 1>
<'s, 1>
<sprung, 1>
<That, 1>

MAP 3:
<like, 1>
<Love, 1>
<melodie, 1>
<my, 1>
<O, 1>
<'s, 1>
<the, 1>

MAP 4:
<in, 1>
<played, 1>
<'s, 1>
<sweetly, 1>
<That, 1>
<tune, 1>
```

```
REDUCE 1 (a..l):
<a, 1>          => <a, 1>
<in, (1,1)>     => <in, 2>
<June, 1>       => <June, 1>
<like, (1,1)>   => <like, 2>
<Love, (1,1)>   => <Love, 2>

REDUCE 2 (m..r):
<melodie, 1>    => <melodie, 1>
<my, (1,1)>     => <my, 2>
<newly, 1>      => <newly, 1>
<O, (1,1)>      => <O, 2>
<played, 1>     => <played, 1>
<red, (1,1)>    => <red, 2>
<rose, 1>       => <rose, 1>

REDUCE 3 (s..z):
<'s, (1,1,1,1)> => <'s, 4>
<sprung, 1>     => <sprung, 1>
<sweetly, 1>    => <sweetly, 1>
<That, (1,1)>   => <That, 2>
<the, 1>        => <the, 1>
<tune, 1>       => <tune, 1>
```

Example 2: Inverted Web Graph

Para cada página, genere una lista de enlaces entrantes.

¿Por qué querrías tener un gráfico así?

Input: Web documents

Map: For each link L in document D emit <href(L), D>

Reduce: Combinar todos los documentos en una lista

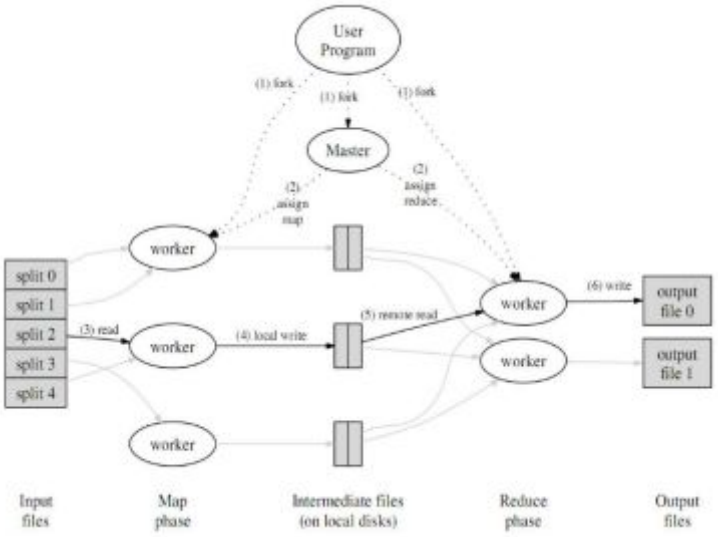
```
techcrunch.com -> apple.com, microsoft.com, ubuntu.com, google.com
reddit.com -> en.wikipedia.org, techcrunch.com
digg.com -> techcrunch.com, microsoft.com, ubuntu.com
```

```
MAP 1:
<apple.com, techcrunch.com>
<microsoft.com, techcrunch.com>
<ubuntu.com, techcrunch.com>
<google.com, techcrunch.com>

MAP 2:
<en.wikipedia.org, reddit.com>
<techcrunch.com, reddit.com>

MAP 3:
<techcrunch.com, digg.com>
<microsoft.com, digg.com>
<ubuntu.com, digg.com>
```

```
REDUCE 1:
<apple.com, techcrunch.com>

REDUCE 2:
<microsoft.com,
   (techcrunch.com, digg.com)>

REDUCE 3:
<ubuntu.com, (techcrunch.com,
   digg.com)>

REDUCE 4:
<google.com, techcrunch.com>

REDUCE 5:
<en.wikipedia.org, reddit.com>

REDUCE 6:
<techcrunch.com, (reddit.com,
   digg.com)>
```

More Examples

Coincidencia de patrones distribuidos
Orden distribuido
La probabilidad de que una palabra esté en mayúscula
Count of URL Access Frequency Term-Vector per Host
Agrupación de documentos
Traducción automática estadística



Functionating

Scheduling

Una maestra, muchas trabajadoras

- Input data split into M map tasks (typically 64 MB in size)
- Reduce phase partitioned into R reduce tasks
- Tasks are assigned to workers dynamically

Maestra asigna cada tarea de mapa a una trabajadora libre

- Considers locality of data to worker when assigning task
- Worker reads task input (often from local disk!)

- Worker produces R local files containing intermediate k/v pairs

Master assigns each map task to a free worker

- Worker reads intermediate k/v pairs from map workers
- Worker sorts & applies user's Reduce operation to produce the output

Localidad

Master program divides up tasks based on location of data: tries to have map tasks on same machine as physical file data, or at least same rack

map task inputs are divided into 64 MB blocks: same size as Google File System chunks

Fault Tolerance

Workers send heartbeats to master

If worker fails

Re-execute completed and in-progress map tasks

Re-execute in-progress reduce tasks

Master notices particular input key/values cause crashes in map, and skips those values on re-execution.

Effect: Can work around bugs in third-party libraries

Master writes checkpoints periodically to database. If master fails a new master is started.

Master is a single machine only, so failing is unlikely, easier just to restart the whole MapReduce task

Task Granularity

Map phase is split on M tasks

Reduce phase is split on R tasks

Practical bounds on how large M and R can be:

Scheduling decisions to make? O(M+R)

States in memory? O(M*R)

Separate output files? R

Example from Google:

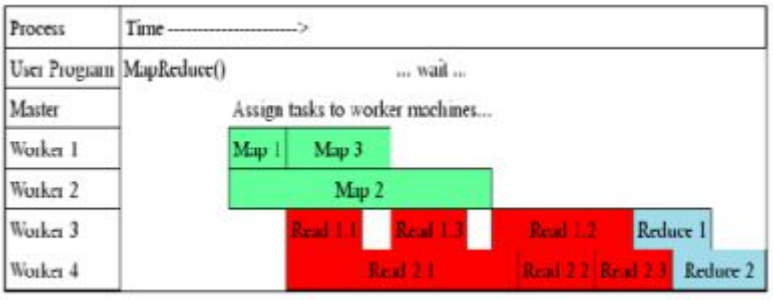
M = 200 k; R = 5 k; workers = 2 k

Fine granularity tasks: many more map tasks than machines

Minimizes time for fault recovery

Can pipeline shuffling with map execution

Better dynamic load balancing



Backup Task

The worst case is not when machine is dead, but when it is barely working, really slow.

Slow workers significantly lengthen completion time

Other jobs consuming resources on machine

Bad disks with soft errors transfer data very slowly

Solution: Near end of phase, spawn backup copies of tasks

Whichever one finishes first "wins"

Effect: Dramatically shortens job completion time

Partitioning

R tasks of reducer should be approximately evenly loaded.

How to ensure this?

Default function uses hashing:

hash(key) mod R

Programmer can override partitioning function

Combiner

"Combiner" function can run on same machine as a mapper

Causes a mini-reduce phase to occur before the real reduce phase, to save bandwidth

Typically has the same code as the reducer

Applicable if a reducer function is

commutative

associative

Chaining MapReduce Executions

Output of a MapReduce task can be processed further as input to another MapReduce task

Examples?

Results from the first reducer may either be written to the permanent storage, or left on reducer machines as temporary files.

Mappers of the second MapReduce job start on the same machines, where reducers of the first one were.

MapReduce Implementations

Patented by Google

Google implementation in C++ with bindings to Python and Java via interfaces, not available publicly

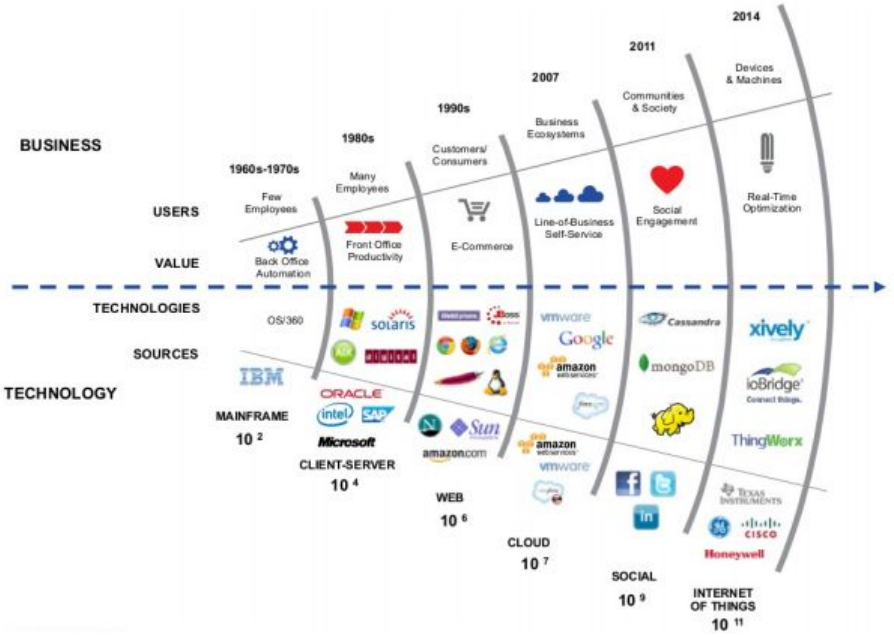
Hadoop project has free open-source implementation in Java

MongoDB

CouchDB

Diapos Hadoop

Contexto histórico



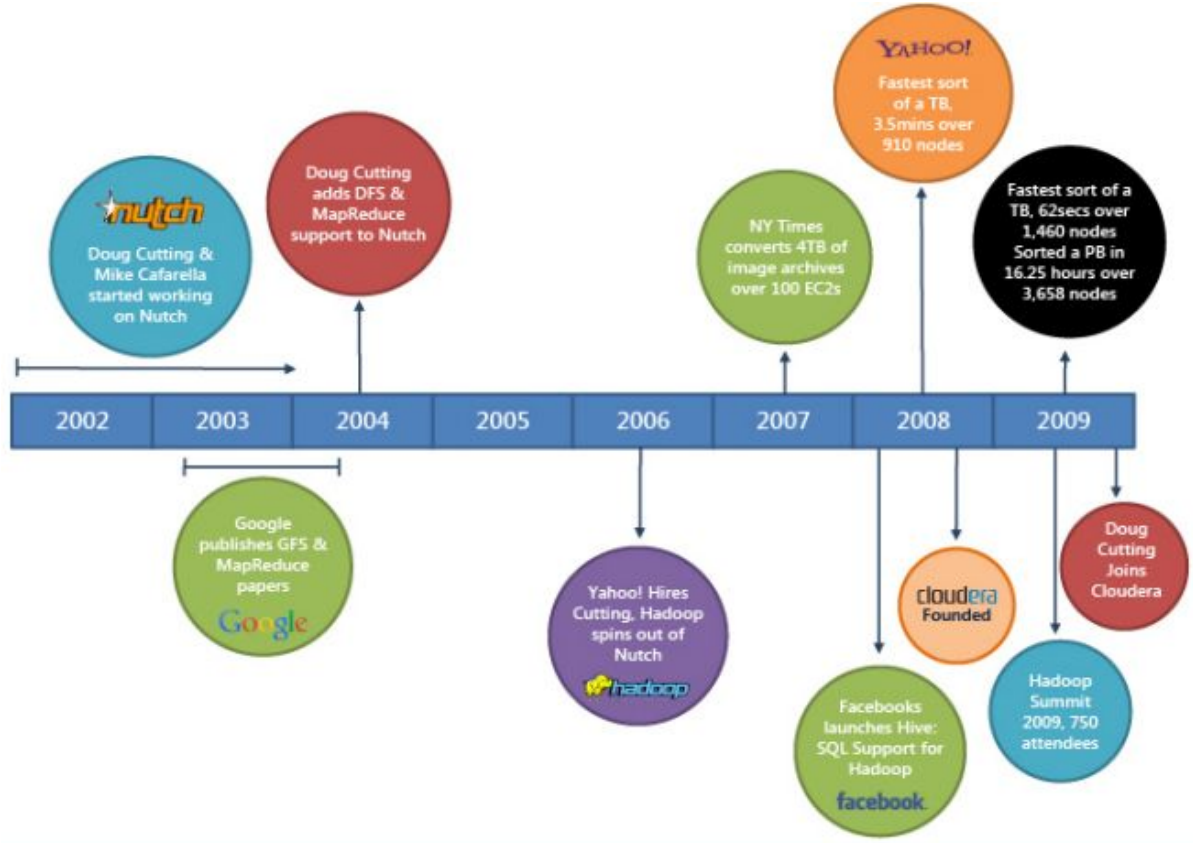
3Vs

Volumen: Grandes cantidades de datos que puden ser empresariales o caracter general

Variedad: Diversidad de fuentes de datos, RRSS, feeds, multimedia, etc

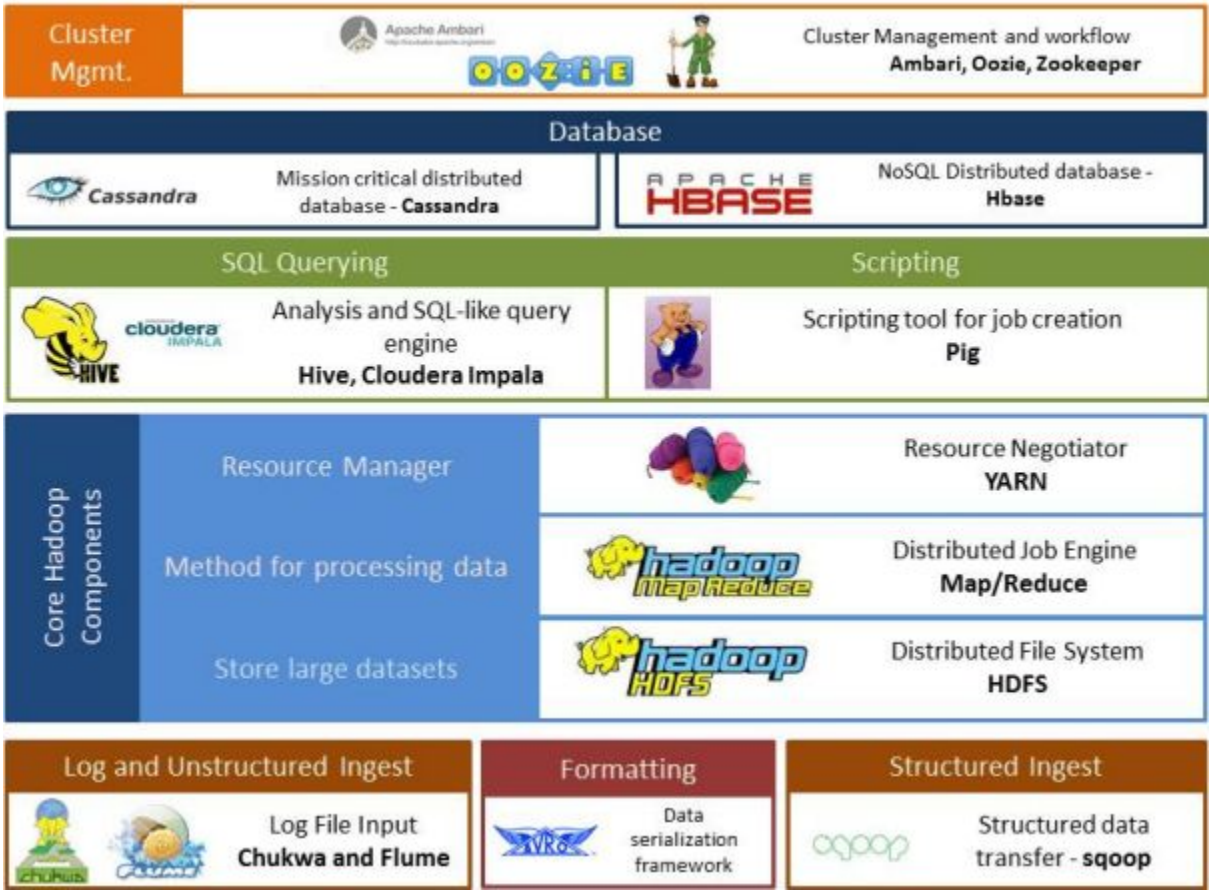
Velocidad: Velocidad de acceso y procesamiento

Historia



Ecosistema Hadoop

- Ambari: Simplificación de la gestión de Hadoop (Clústeres).
- Avro: Serialización de datos.
- Cassandra: Base de datos NoSQL distribuida.
- Chukwa: Análisis y recolección de Logs.
- HBase: Base de datos NoSQL Distribuida.
- Hive: Lenguaje de alto nivel similar a SQL que se traduce en tareas MapReduce.
- Mahout: Librería de aprendizaje automático y minería de datos.
- Pig: Lenguaje de alto nivel para generación de tareas MapReduce.
- Spark: Motor de cálculo general.
- Tez: Framework de programación de flujos de datos de carácter general.
- ZooKeeper: Servicio de coordinación distribuido de Hadoop.
- Oozie: Flujo de trabajo para tareas MapReduce
- Sqoop: Conectividad entre bases de datos tradicionales y Hadoop.
- Flume: Transferencia de datos entre los sistemas de empresa y Hadoop



Core Hadoop

Sistema de ficheros distribuido auto gestionado + Sistema de computación distribuido con tolerancia a errores y abstracción de computación paralela

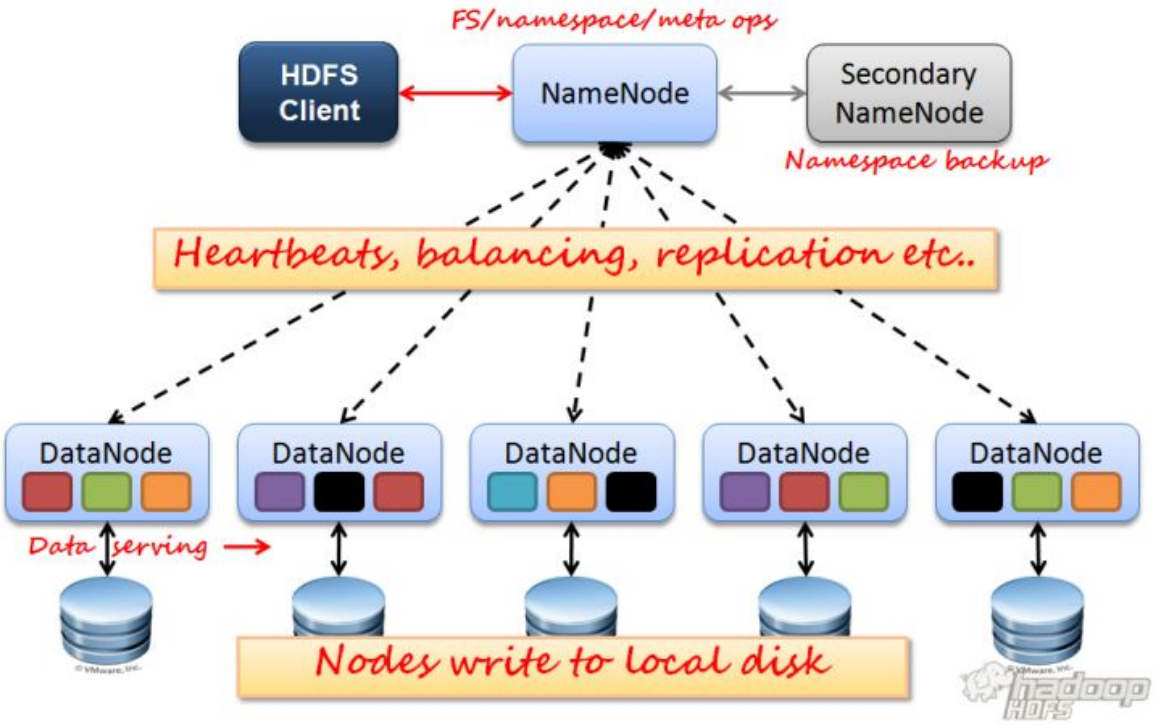
HDFS

HDFS (Hadoop Distributed File System) es una evolución de GFS (Google File System). Es un sistema de archivos distribuido diseñada para contener gran cantidad de datos y el acceso concurrente a los mismos.

HDFS - Características

- Capacidad de almacenaje de grandes cantidades de datos (Terabytes o Petabytes)
- Fiabilidad en el almacenamiento en Cluster
- Posibilidad de leer los ficheros localmente
- Diseñada para lecturas intensas (Penalización sobre la búsqueda aleatoria)
- Operaciones
    - Lectura
    - Escritura
    - Borrado
    - Creación

HDFS - Funcionamiento

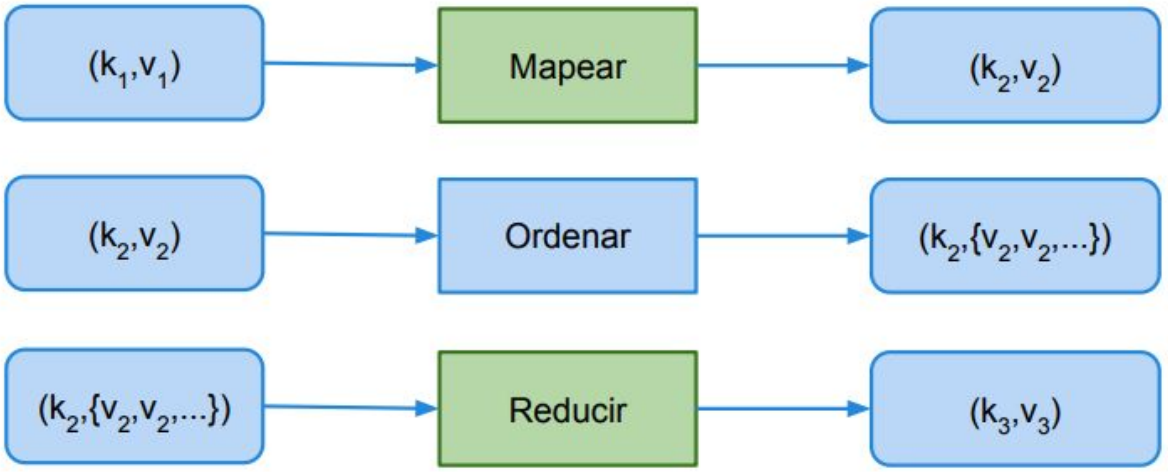


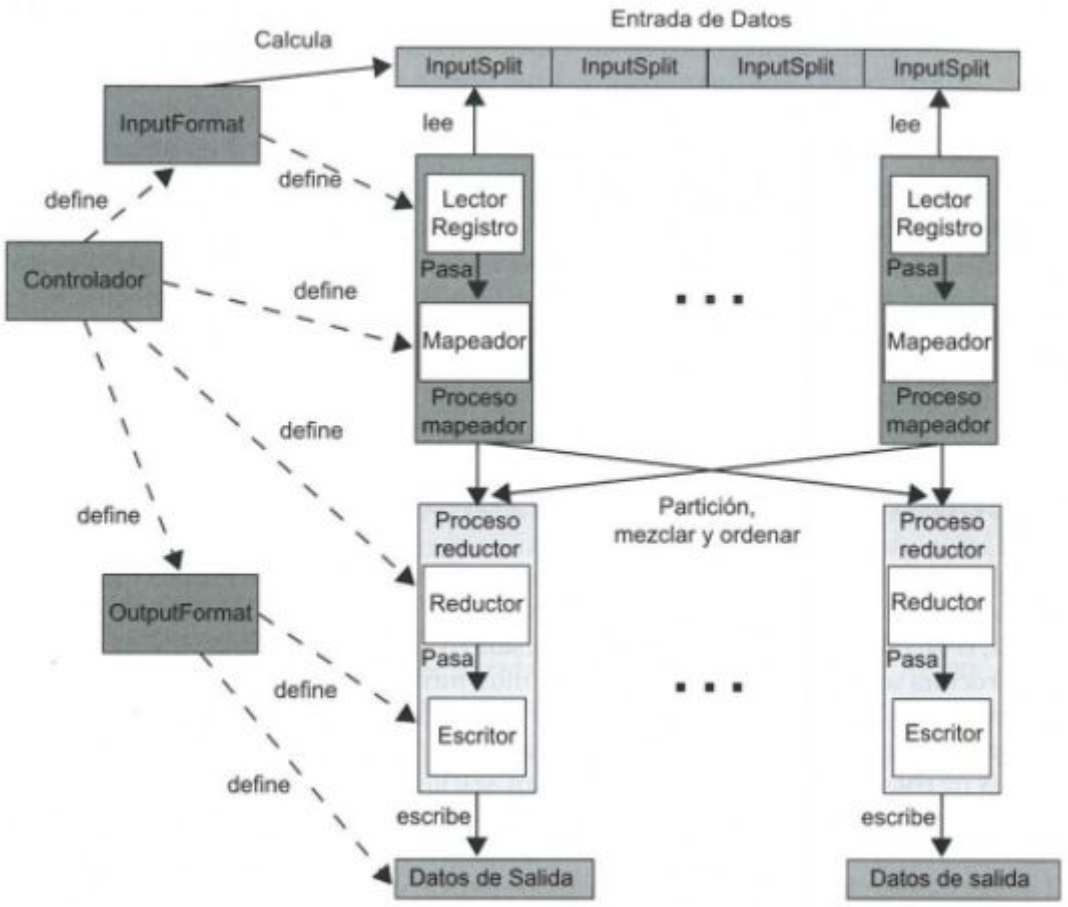
MapReduce

Estructura para ejecutar algoritmos altamente paralelizables y distribuidos utilizando ordenadores "básicos"

Origen del concepto de combinador Map y Reduce de lenguajes de programación funcional como Lisp.

Divide y vencerás





Controlador

- Inicializa el trabajo MapReduce
- Define la configuración
- Enumera los componentes
- Monitoriza la ejecución

Datos de entrada

- Ubicación de los datos para el trabajo
- HDSF
- HBase
- Otros

InputFormat

- Definición de cómo se leen y dividen los datos.
- Definición de InputSplit
- Definición de número de tareas Map

InputSplit

- Define la unidad de trabajo para una tarea Map única

RecordReader

- Define un subconjunto de datos para una tarea map
- Lee los datos desde el origen
- Convierte los datos en pares clave / valor

Mapeador

- Ejecuta un algoritmo definido por el usuario.
- Instancias JVM por tarea Map.
- Aislamiento de tareas
- Fiabilidad del resultado dependiente únicamente de la máquina local.

Partición

- Elección de dónde reducir cada par clave / valor

Mezclar

- Las tareas Reduce trabajan sobre la misma clave
- Aquí se desplazar las salidas de las tareas Map a donde se necesite

Ordenar

- Se reciben los pares clave / valor Mezclados anteriormente y se ordenan para pasarlos al reductor

Reductor

- Ejecuta un algoritmo definido por el usuario
- Recibe una clave y todos los valores asociados

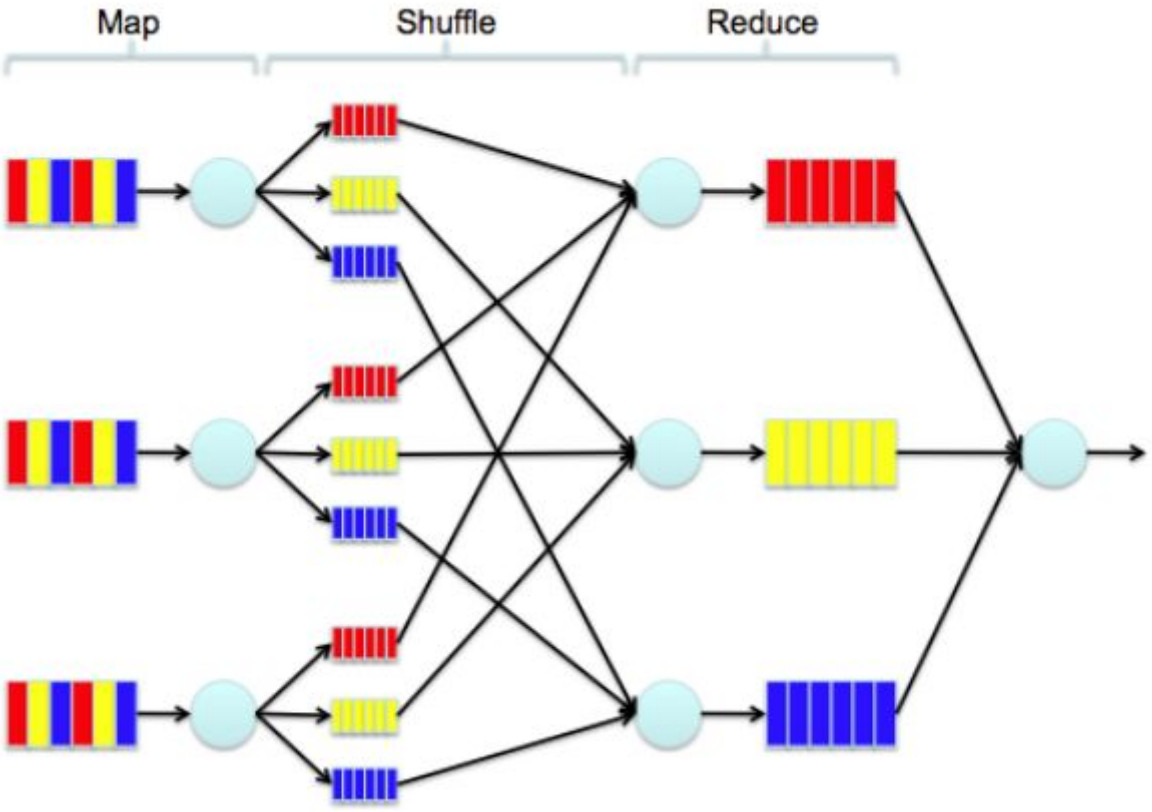
OutputFormat

- Define cómo se escribe la salida del proceso MapReduce.
- Define la ubicación del RecordWriter
- Definir los datos a devolver

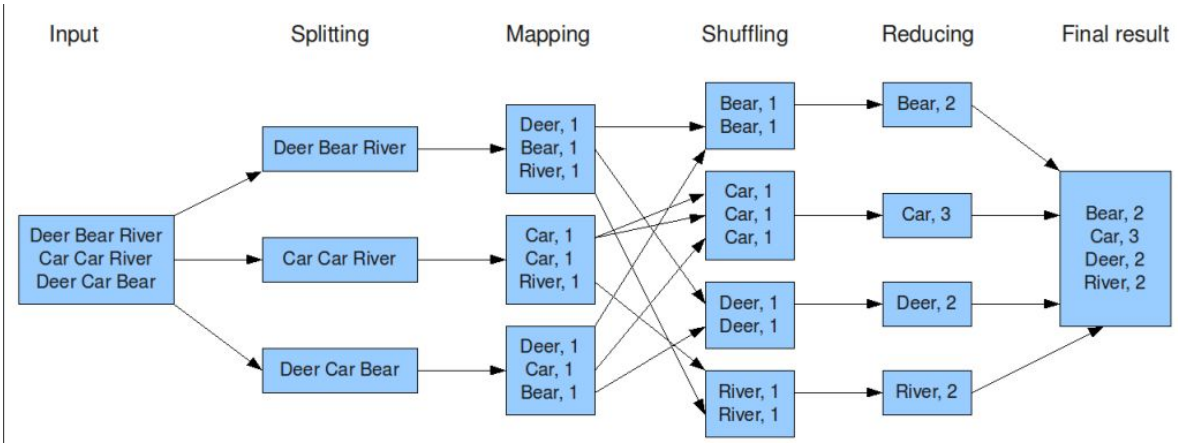
RecordWriter

- Define cómo se escriben los registros individuales de salida

MapReduce: Diagrama de Nodos



MapReduce - Ilustración

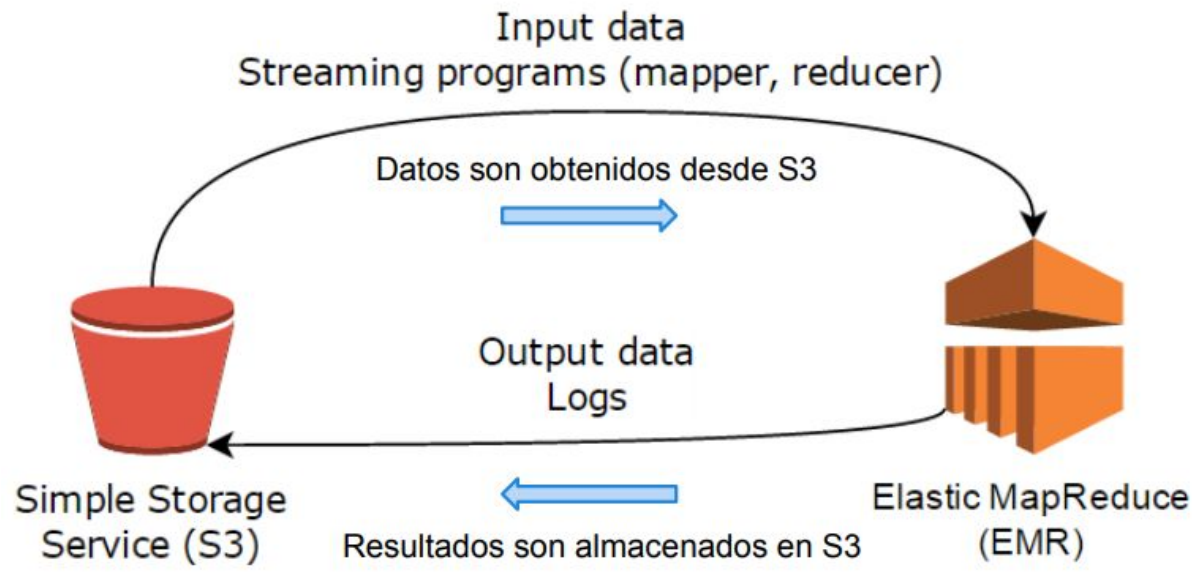


Elastic Map Reduce

Servicio Web de AWS (Amazon Web Services) para implementar el MapReduce mediante Apache Hadoop.

Permite crear trabajos en Apache Hadoop que operan sobre datos en Amazon S3 en instancias de Amazon EC2

Elastic Map Reduce - Arquitectura



Elastic Map Reduce - Streaming

Streaming consiste en analizar y aplicar MapReduce mediante el uso de un Mapper (cuya función es Organizar) y un Reducer (cuya función es Agregar).

Un Streaming job (tarea) es una tarea de Hadoop que consiste de un Mapper y Reducer. Los Mapper y Reducer pueden ser escritos en: Java, Ruby, PHP, Perl, Python, R, Bash, C++..

1) Crear el stream job en la Consola de AWS.
2) Indicar parámetros del stream job
3) Configurar instancias EC2
4) Lanzar el job
5) Revisar ejecución
6) Revisar resultados

“Mapper”

```
#!/usr/bin/php
<?php
//sample mapper for hadoop streaming job
$word2count = array();

// input comes from STDIN (standard input)
while (($line = fgets(STDIN)) !== false) {
  // remove leading and trailing whitespace and lowercase
  $line = strtolower(trim($line));
  // split the line into words while removing any empty string
  $words = preg_split('/\W/', $line, 0, PREG_SPLIT_NO_EMPTY);
  // increase counters
  foreach ($words as $word) {
     $word2count[$word] += 1;
  }
}

// write the results to STDOUT (standard output)

foreach ($word2count as $word => $count) {
  // tab-delimited
  echo "$word\t$count\n";
}

?>
```

“Reducer”

```
#!/usr/bin/php
<?php
//reducer script for sample hadoop job
$word2count = array();

// input comes from STDIN
while (($line = fgets(STDIN)) !== false) {
   // remove leading and trailing whitespace
   $line = trim($line);
   // parse the input we got from mapper.php
   list($word, $count) = explode("\t", $line);
   // convert count (currently a string) to int
   $count = intval($count);
   // sum counts
   if ($count > 0) $word2count[$word] += $count;
}

ksort($word2count);  // sort the words alphabetically

// write the results to STDOUT (standard output)
foreach ($word2count as $word => $count) {
   echo "$word\t$count\n";
}

?>
```

Showtime

Vamos a lanzar 2 ejecuciones reales de EMR: ● Con 1 Nodo ● Con 2 Nodos

Aplicaciones de Hadoop

● Detección de fraude bancario.
● Análisis de Marketing en redes sociales
    ○ Twitter genera 12 TB de información al día
● Análisis de patrones de compra
    ○ Wallmart utilizaba en 2012 30.000 millones de sensores RFID
● Reconocimiento de patrones de tráfico para el desarrollo urbano
● Previsión de averías en aviones ○ El airbus A380 genera 640 TB de información por vuelo
● Transformación de datos grandes
● Aplicación de algoritmos de reconocimiento facial ○ En 2012, facebook publicó que se suben 250 millones de fotos al día

Que no es hadoop

Una base de datos
La solución a todos los problemas
HDFS no es un sistema de archivos POSIX completo

¿Donde aplicaríais Hadoop?

● Procesamiento paralelo
● Sistema de ficheros distribuido
● Heterogeneidad de fuentes
● Tamaño de las fuentes
● Dimensionamiento de las infraestructuras

Casos de éxito - Análisis de Riesgos

Reto

Con la crisis de 2008, una importante entidad financiera quedó expuesta a la morosidad de los clientes, era vital mejorar los análisis de riesgo.

Solución

● Creación de un cluster único con Hadoop (Con petabytes de información)
● Cargo la información de todos los almacenes de datos de la entidad que disponían de una visión específica del cliente
● Cargo información no estructurada
    ○ Chats
    ○ Correos al servicio de atención al cliente
    ○ Registros de los Call Center
    ○ Otros orígenes
● Capacidad para realizar un análisis completo de la situación de los clientes

Casos de éxito - Fuga de clientes

Reto

Una importante compañía de telecomunicaciones necesitaba entender porque perdía clientes, para ello, necesitaba dar respuesta a las siguientes preguntas: ¿Eran clientes que se iban o simplemente estaban negociando las condiciones? ¿Se iban a la competencia? ¿Se iban por problemas en la cobertura? ¿Por los precios? ¿Por incidencias en los dispositivos? ¿Por otros motivos?

Solución

● Creación de un cluster único con Hadoop
● Se combinaron las fuentes transaccionales tradicionales y las redes sociales
● Analizaron los registros de llamadas y crearon una red de contactos de los clientes
● Cruzaron esta información con los contactos de las fugas en las redes sociales y concluyeron que cuando un cliente se iba de la compañía, sus contactos eran más proclives a abandonar también.

● Cruzaron los mapas de cobertura con la ubicación de los cliente y dimensionan el impacto de las incidencias de cobertura en la fuga de clientes
● Optimizaron la inversión en infraestructuras y el desarrollo de nuevos productos

Casos de éxito - Puntos de venta

Reto

Una importante empresa de Retail quería incorporar las nuevas fuentes de información disponibles en los análisis (Tiendas Online, Tiendas Offline, Redes Sociales...). Los sistemas tradicionales son muy caros para almacenar datos complejos.

Solución

● Creación de un cluster único con Hadoop
● Cargaron 20 años de transacciones
● Utilizaron Hive para realizar algunos de los análisis que ejecutaban realizaban en el almacén de datos, aunque extendiéndose a periodos mucho mayores.
● Redujeron los costes de infraestructura
● Incluyeron nuevos orígenes de datos (Canales de noticias, Redes Sociales, Twitter...)

Casos de éxito - Datos en bruto

Reto

Una importante agencia de viajes genera volúmenes de datos en bruto enormes que solo podían almacenar unos días por el coste del almacén de datos.

Solución

● Creación de un cluster único con Hadoop
● Almacenarón toda la información a un coste muy inferior al guardarlas en el formato original y comprimida
● Utilizaron Hive para analizar el comportamiento en la web de reservas.
● Consiguieron un mejor entendimiento de sus clientes y pudieron ofrecen mejores productos aumentando la rentabilidad.

CAPITUO 4 Arquitectura Lambda

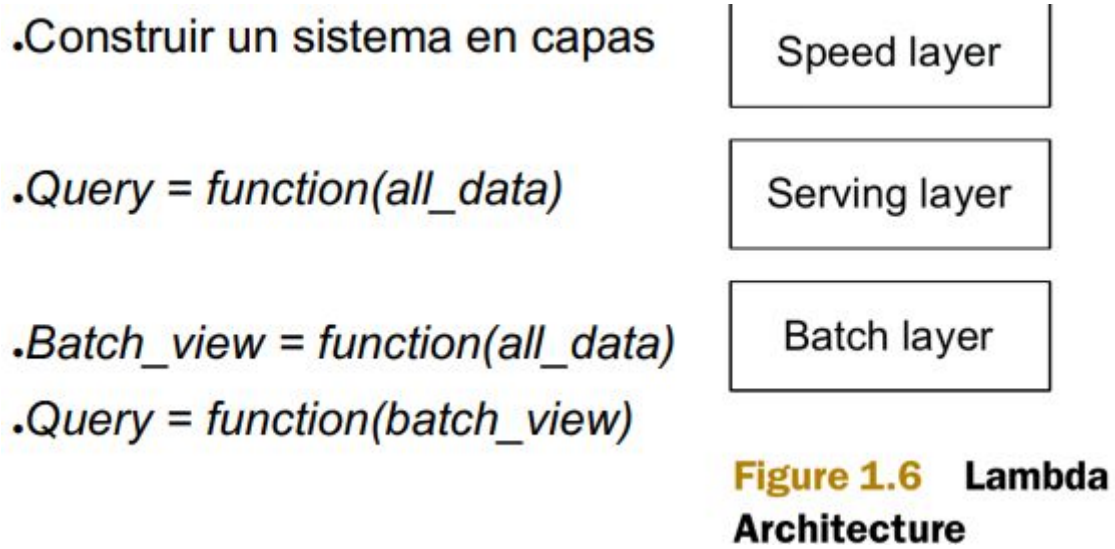


Figure 1.6 Lambda Architecture

●Número de visitas a una página web en un rango de fechas

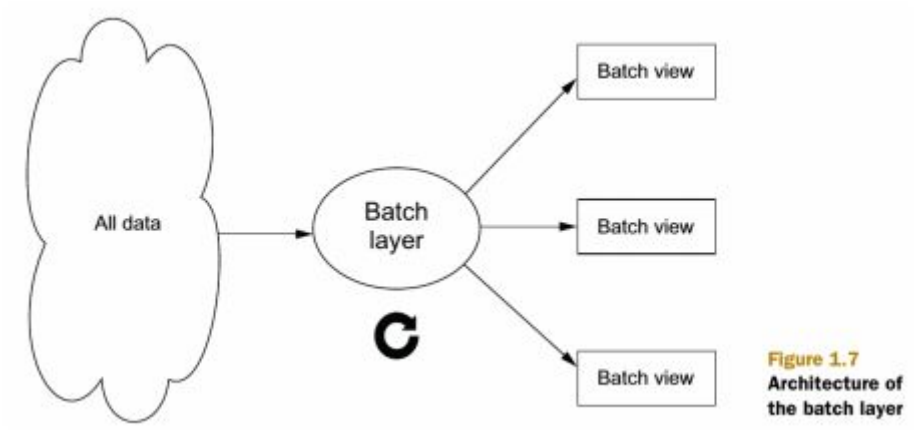


Figure 1.7 Architecture of the batch layer

Batch Layer

- Batch_view = function(all_data)
- Copia maestra del dataset (todos los datos)
- Precalcula las vistas arbitrarias sobre ese dataset

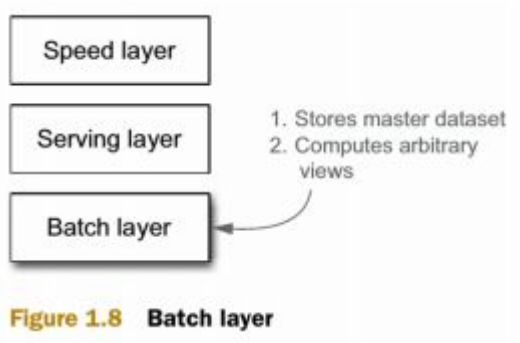


Figure 1.8 Batch layer

Serving Layer

- La capa Batch emite vistas como resultado
- El objetivo es almacenar esas vistas en algún lugar para que puedan ser consultadas
- Capa de servicios
    - Actualizaciones desde la capa batch.
    - Lecturas randómicas
- No necesita soportar escrituras randómicas
- La escrituras randómicas aumentan la complejidad
- Simplicidad: robusta, predecible, fácil de configurar y fácil de operar

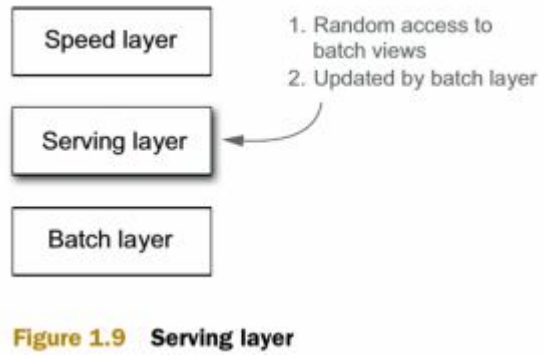


Figure 1.9 Serving layer

Batch and Serving Layer

- Dan soporte a queries arbitrarios sobre un cualquier dataset.
- Tienen un desface de unas cuantas horas

●Satisfacen todas las propiedades deseadas de un sistema de Big Data.

- ● Robustez y tolerancia a fallos:
  - ● Hadoop maneja la recuperación de errores cuando se cae algún servidor
  - ● La capa de servicios usa replicación para asegurar la disponibilidad.
  - ● Permite corregir errores humanos ya que se pueden recalcular las vistas desde cero

●Satisfacen todas las propiedades deseadas de un sistema de Big Data.

- ● Escalabilidad:
  - ● Ambas se manejan con sistemas distribuidos fáciles de escalar tan solo añadiendo nuevos servidores al cluster.
- ● Generalización:
  - ● La arquitectura descrita permite calcular vistas arbitrarias sobre cualquier dataset.

Batch and Serving Layer

●Satisfacen todas las propiedades deseadas de un sistema de Big Data.

- ● Extensibilidad:
  - ● Agregar una nueva vista es simplemente añadir una nueva función sobre el dataset maestro.
- ● Consultas Adhoc
  - ● Toda la data está disponible en una sola ubicación, haciendo sencillo ejecutar cualquier query sobre dicha data.

●Satisfacen todas las propiedades deseadas de un sistema de Big Data.

- ● Mantenimiento mínimo:
  - ● Hadoop requiere cierto conocimiento especializado pero es bastante simple de configurar y operar.
  - ● La capa de servicios al no requerir escrituras randómicas se vuelve mucho más simple de mantener.
- ● Lo único que falta es cómo manejar la baja latencia de los updates.

Speed Layer

●En esta capa se procesan los datos nuevos que ingresan al sistema cuando se están calculando las vistas.

●El procesamiento es en tiempo real.

●Realtime View = function(realtime_view, new data)

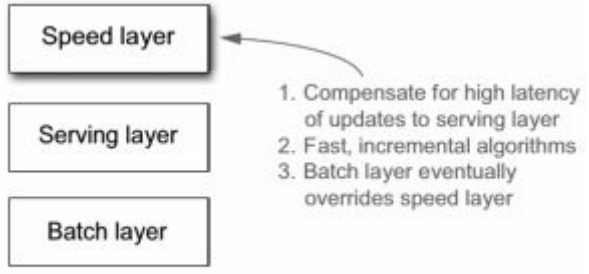


Figure 1.10 **Speed layer**

Sistema de Big Data

●Batch View = function(all data)

●Realtime View = function(realtime_view, new data)

●Query = function(batch view, realtime view)

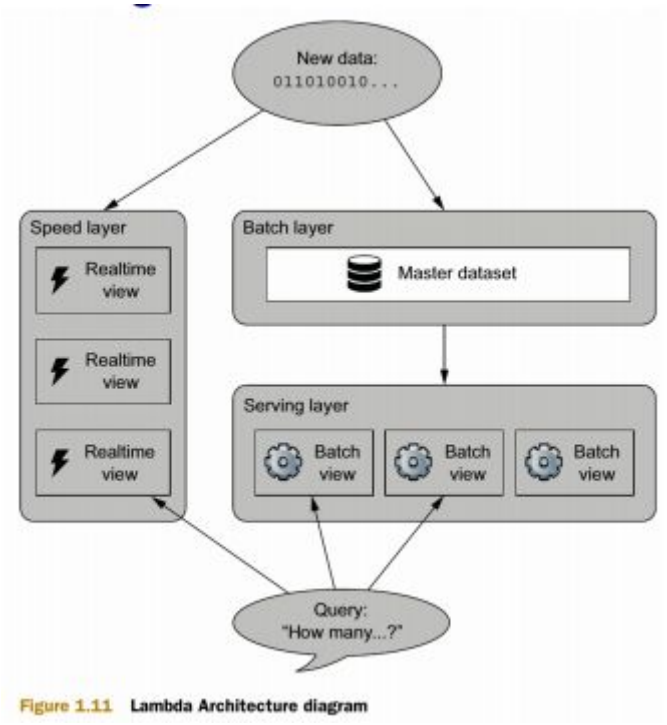


Figure 1.11 **Lambda Architecture diagram**

- Propiedades de los queries
  - Latencia: El tiempo que le toma ejecutarse
  - Timeliness: Que tan actualizados son los resultados de la consulta
  - Exactitud: Aproximaciones a los resultados de las consultas para ganar rendimiento y escalabilidad

Detalles de la capa Batch

- Una buena métrica es cuánto tiempo toma actualizar las vistas
- Mientras más tiempo le tome a la capa Batch pre calcular las vistas
  - Más grande debe ser la capa en tiempo real.
  - Más tiempo toma la recuperación ante errores de programación (bugs)
- Procesamiento en Batch Incremental

Procesamiento en Batch Incremental

- Algoritmos Incrementales
- Algoritmos de re-cálculo
- Contar el número total de registros del master dataset

.Algoritmos de re-cálculo

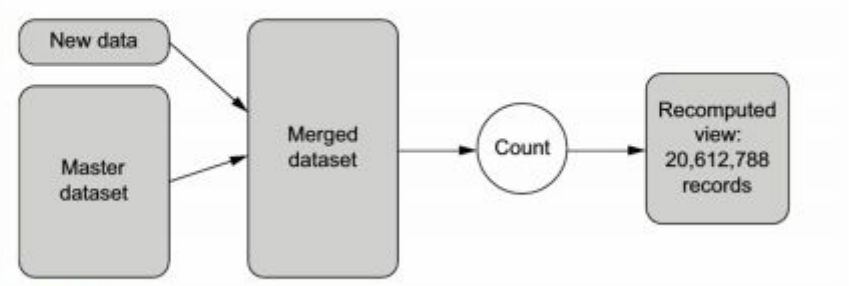


Figure 6.5 **A recomputing algorithm to update the number of records in the master dataset. New data is appended to the master dataset, and then all records are counted.**

.Algoritmos Incrementales

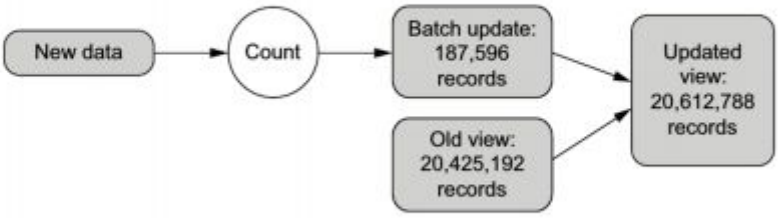


Figure 6.6 An incremental algorithm to update the number of records in the master dataset. Only the new dataset is counted, with the total used to update the batch view directly.

●Algoritmos Incrementales vs. re-cálculo

Table 6.1 Comparing recomputation and incremental algorithms

	Recomputation algorithms	Incremental algorithms
Performance	Requires computational effort to process the entire master dataset	Requires less computational resources but may generate much larger batch views
Human-fault tolerance	Extremely tolerant of human errors because the batch views are continually rebuilt	Doesn't facilitate repairing errors in the batch views; repairs are ad hoc and may require estimates
Generality	Complexity of the algorithm is addressed during precomputation, resulting in simple batch views and low-latency, on-the-fly processing	Requires special tailoring; may shift complexity to on-the-fly query processing
Conclusion	Essential to supporting a robust data-processing system	Can increase the efficiency of your system, but only as a supplement to recomputation algorithms

Medir y optimizar el rendimiento en la capa Batch

●Después de duplicar el tamaño del cluster la latencia bajó de 30 horas a 6 horas. (80% de ganancia)

●Después de reconfigurar mal un cluster de Hadoop, se tenía 10% más de fallas de los servidores. Esto incrementó la latencia de 8 horas a 72 horas.

●T = O + PH

T: es el tiempo de ejecución en horas

O – (Sobrecarga): es el tiempo independiente de los datos a procesar en horas. Configurar los procesos, copiar los datos en el cluster, etc.

H: es el número de horas de datos procesadas en esa iteración

P: El tiempo de procesamiento dinámico. El número de horas que cada hora de datos agrega al tiempo total. Si cada hora de datos agrega media hora al total, P¿0.5

$$T = O + PT$$

$$T = \frac{O}{(1-P)}$$

●H variará en cada iteración dependiendo si la iteración anterior tomo más o menos tiempo.

●Para determinar cuando el tiempo de procesamiento se estabiliza se debe considerar cuando el tiempo total (T) es igual al número de horas de datos que procesa (H)

●El tiempo total es directamente proporcional a la sobrecarga

●El tiempo total no es directamente proporcional al tiempo de procesamiento dinámico.

$$T = O + PT$$

$$T = \frac{O}{(1-P)}$$

●Qué pasa si P es mayor o igual que 1?

Cada iteración tendrá más datos que la anterior.

El procesamiento estará retrasado siempre.

●Al incrementar el tamaño del cluster al doble, P se reduce en aproximadamente la mita

$$T_1 = \frac{O}{(1-P)}$$

$$T_2 = \frac{O}{\left(1-\frac{P}{2}\right)}$$

●6 min, no se gana demasiado

●54 min, la ganancia es bastante considerable

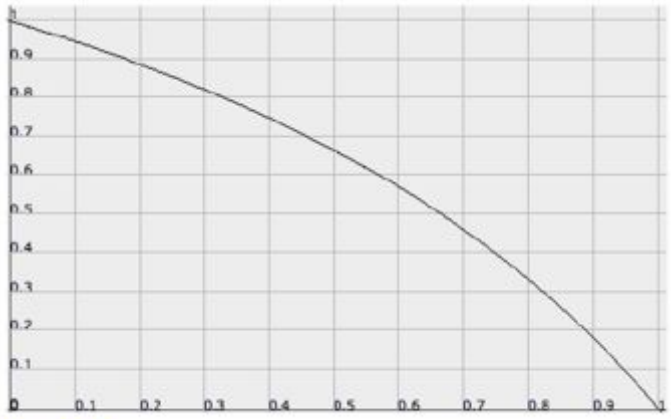


Figure 18.3 **Performance effect of doubling cluster size**

●10% de fallos

– Si se tienen 100 tareas 10 fallan y se tienen que reintentar

– De las 10 fallará 1, que también se reintentará

– P aumentará en un 11%

$$T_1 = \frac{O}{(1-P)}$$

$$T_2 = \frac{O}{(1-1.11P)}$$

●P debajo de 0.7

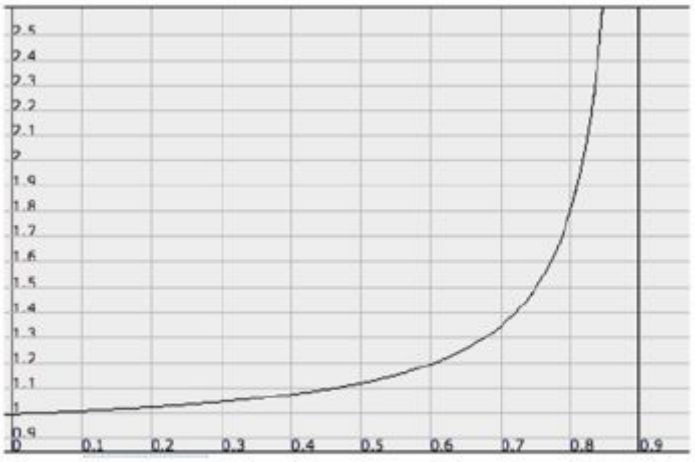


Figure 18.4 **Performance effect of 10% increase in error rates**

- Si P es mayor a 0.5 añadiendo 1% de máquinas reducirá la latencia en más de 1%
- Si P es menor a 0.5 añadiendo 1% de máquinas reducirá la latencia en menos de 1%

Figure 18.4 **Performance effect of 10% increase in error rates**

Detalles de la capa en tiempo real (Speed Layer)

- La capa de servicios se actualiza con una latencia alta.
  - Siempre está desactualizada unas horas
  - Las vistas representan la mayoría de nuestros datos
  - Los únicos datos faltantes son los que ingresaron después de la última actualización de las vistas
- Calcular queries en tiempo real para compensar esa pocas horas de datos.
  - La capa en tiempo real (Speed Layer)
- En esta capa es donde optamos por rendimiento
  - Algoritmos incrementales en lugar de recálculo
  - Bases de datos mutables (lecturas / escrituras)
- Se necesita latencia baja
- Los errores humanos no son un problema La capa de servicios sobreescribe esta capa

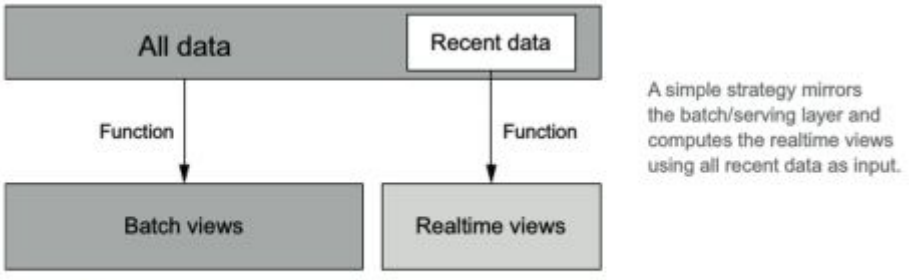


Figure 12.2 **Strategy: realtime view = function(recent data)**

- Este esquema sirve si la aplicación acepta una latencia de unos cuantos minutos

●Usualmente la latencia en esta capa debe estar en el orden de los milisegundos

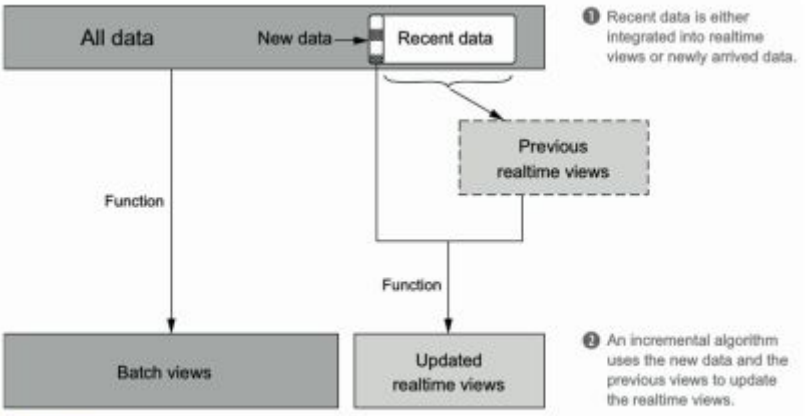


Figure 12.3 **Incremental strategy: realtime view = function(new data, previous realtime view)**

Capa de consultas

- ●Batch y Speed layer para responder consultas
- ●Que usar de cada capa y cómo unir todo para mostrar los resultados correctos