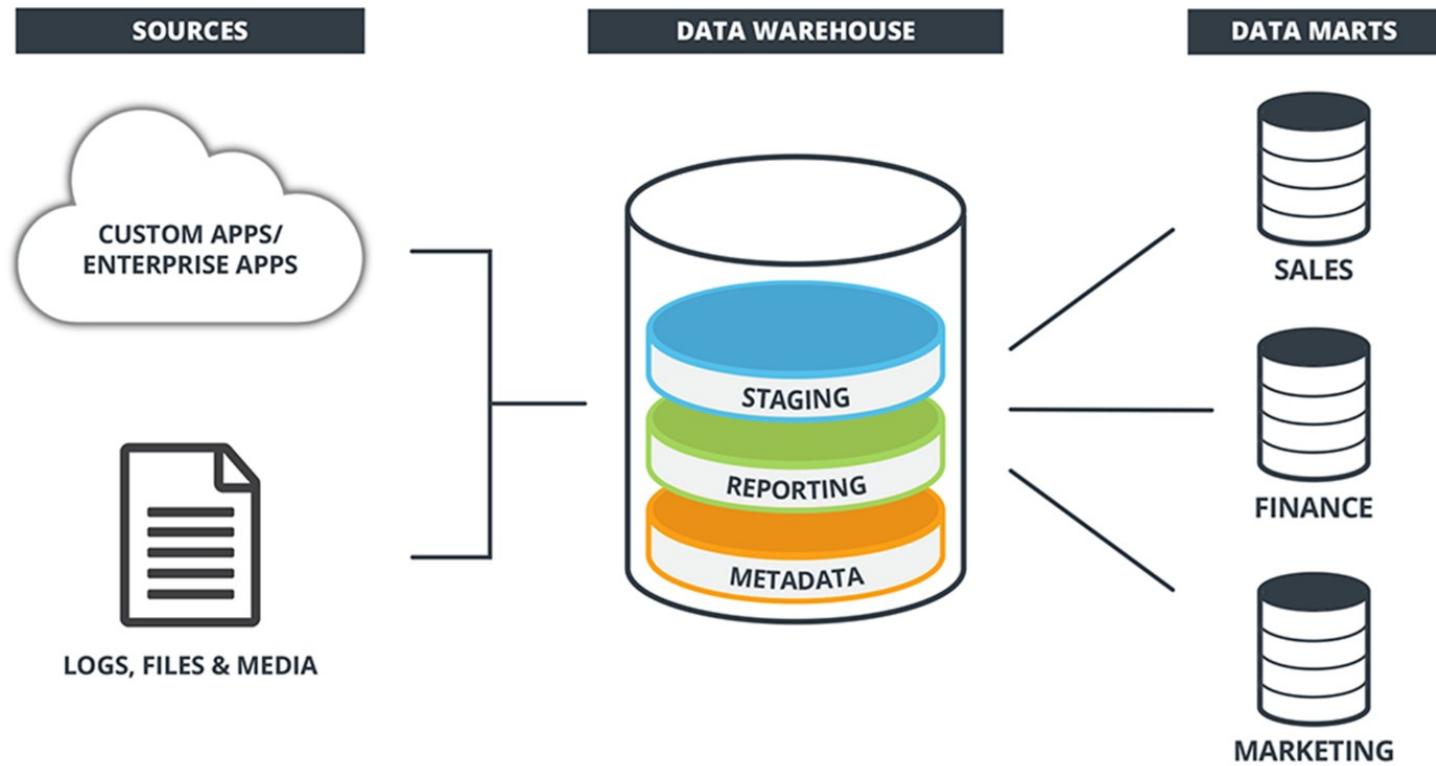


Data Warehouse

DWH 1: <https://claude.ai/chat/8efa22dc-4225-4ec6-98d4-7d82458d8b65>

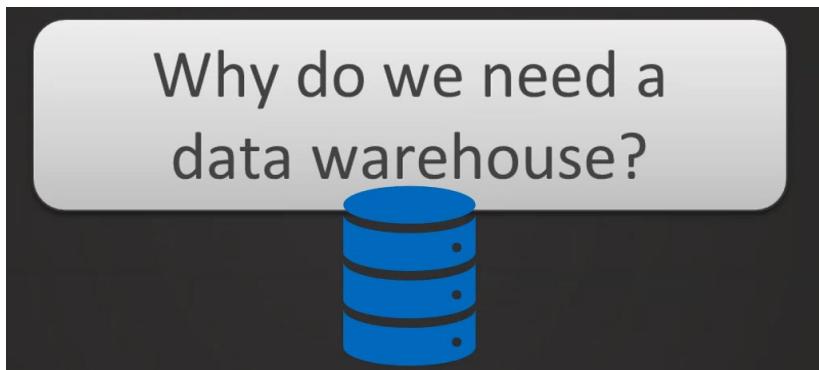
DWH 2: <https://claude.ai/chat/6d9add46-870e-40ef-8034-4081ee5e6238>



1. Data Warehouse Basic

1.1 ¿Por qué un Data Warehouse?

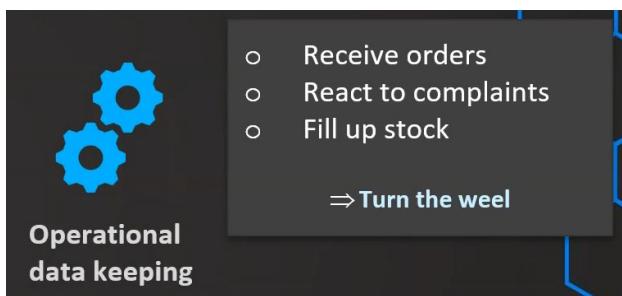
1.1.1 ¿Por qué necesitamos un almacen de datos (data Warehouse) (2 razones)



Para entenderlo y responder a esa pregunta, tenemos que analizar los dos propósitos diferentes de como y por que utilizamos los datos en una empresa.

1) La primera es con fines operativos:

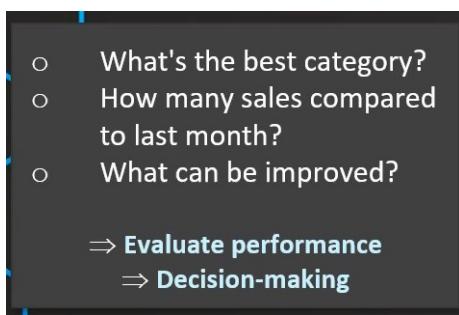
Queremos utilizar los datos para recibir y tramitar pedidos, para recibir reclamaciones y reaccionar ante ellas, para reponer existencias y, para hacer todo lo necesario para que nuestra empresa funcione y la rueda siga girando. Y también para tomar mejores decisiones de cara al futuro y para comprender nuestra empresa



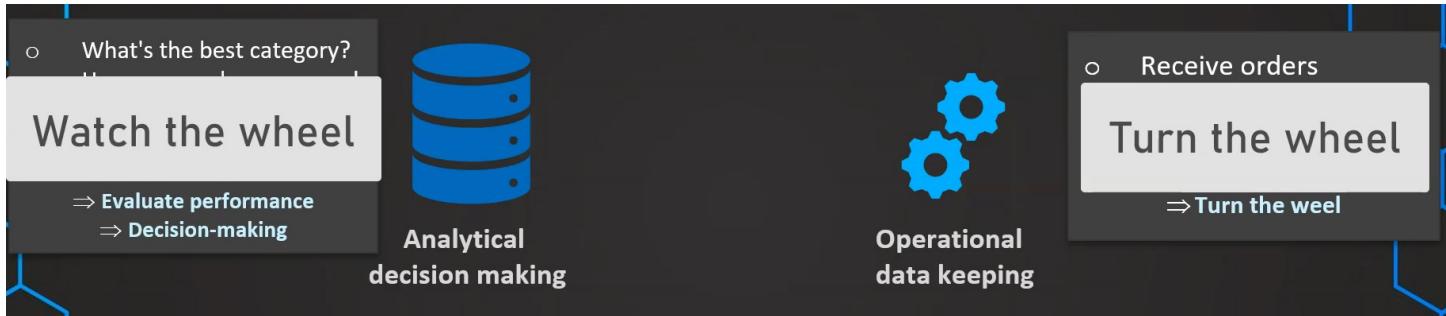
[tenemos los datos operativos para que la rueda siga girando]

Queremos tener preguntas como:

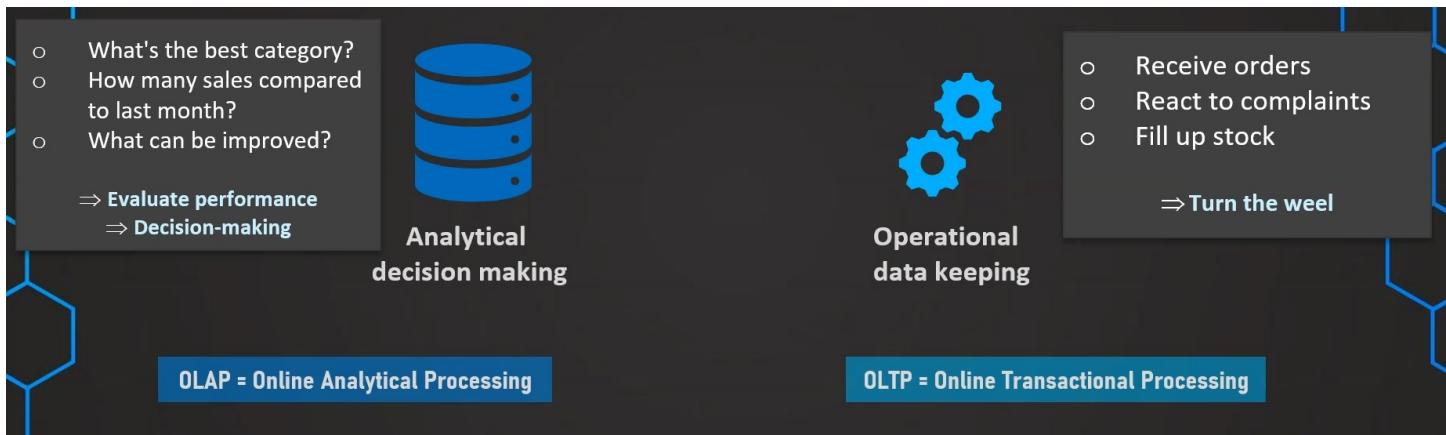
1. ¿Cuál es la mejor categoría en la que estamos vendiendo productos?
2. ¿Cuál es el numero de ventas que tenemos este mes en comparación con el mes pasado?
3. ¿Qué podemos hacer para mejorar las cosas en nuestra empresa?



Los datos operativos nos permiten girar la rueda, y el tratamiento analítico nos permite observar como gira esa rueda y que se puede mejorar en la empresa.



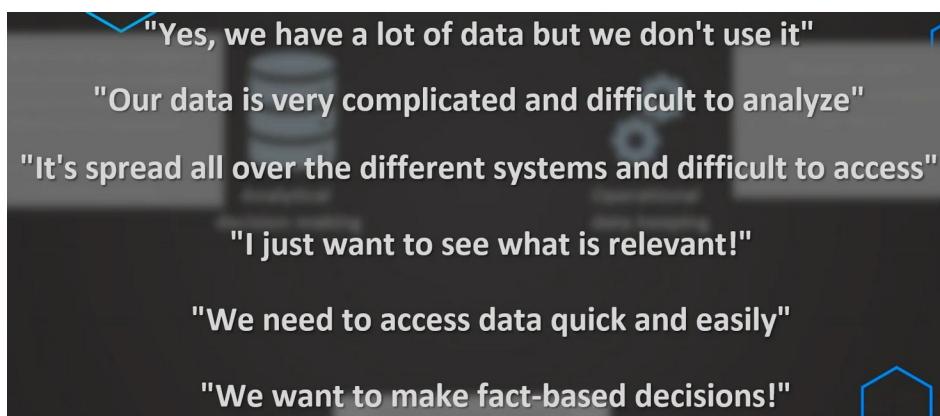
Este mantenimiento de datos operativos también se denomina **OLTP** (procesamiento transaccional en linea) y el procesamiento analítico se denomina **OLAP**



1.1.2 Manifiesto de la falta de un almacén de datos

Se pone de manifiesto en afirmaciones como:

1. Tenemos muchos datos, pero realmente no podemos utilizarlos, o no los utilizamos
2. Muy complicado acceder a ellos, muy difícil analizarlos
3. Están repartidos por diferentes sistemas de nuestra empresa y muy difícil acceder
4. Solo quiero ver lo que es relevante y quiero tener esos datos accesibles de forma rápida y sencilla
5. Queremos tomar decisiones basadas en hechos y no discutir más sobre los números.



1.1.3 Requisitos distintos para los sistemas

Tratamiento de datos operativos: solemos procesar un registro cada vez para que la empresa siga funcionando, y también solemos querer introducir algunos datos o editar otros, y normalmente sólo nos preocupan los datos actuales. Y por esta razón, no solemos guardar un historial muy largo de los datos.

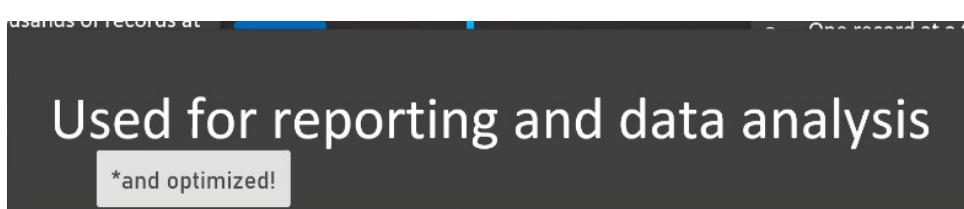
Tratamiento analítico de datos: Aquí analizamos y obtenemos normalmente miles y millones de registros al mismo tiempo. Así, por ejemplo, queremos analizar la media de las ventas de los últimos seis meses, y para obtener esta información rápidamente, es muy importante tener un rendimiento de consulta rápido. Y también queremos dar sentido a estos datos y, por tanto, necesitamos contexto. Así que queremos analizar datos a lo largo del tiempo o a través de múltiples categorías,



y puesto que tenemos esos requisitos tan diferentes, tiene mucho sentido mantener esos **diferentes sistemas separados:** **Y ahora existe un almacén de datos para atender las necesidades de datos analíticos.** Así que esto es básicamente lo que es un almacén de datos,

¡El DWH (Data Warehouse) está ahí para abordar esas necesidades de datos analíticos!

Definición de un almacén de datos: Es una ubicación de datos que se utiliza para la presentación de informes y análisis de datos.



1.1.4 RESUMIENDO: OLTP vs OLAP

OLTP (Online Transaction Processing): Procesamiento de Transacciones en Línea

Sistema diseñado para manejar operaciones diarias en tiempo real. Se enfoca en procesar transacciones individuales de forma rápida y confiable, manteniendo la integridad de los datos.

Propósito: **Hacer que el negocio funcione día a día (operacional).**

OLAP (Online Analytical Processing)

Procesamiento Analítico en Línea: Sistema diseñado para analizar grandes volúmenes de datos históricos. Se enfoca en descubrir patrones, tendencias y generar informes para tomar decisiones estratégicas.

Propósito: **Analizar cómo funciona el negocio y tomar mejores decisiones (analítico).**

1.1.4.1 Diferencias Principales

Característica	OLTP	OLAP
Tipo de operación	Transacciones (INSERT, UPDATE, DELETE)	Consultas complejas (SELECT con agregaciones)
Volumen de datos	Pocos registros por operación	Miles/millones de registros por consulta
Velocidad requerida	Respuesta inmediata (milisegundos)	Puede tomar segundos/minutos
Datos	Actuales y detallados	Históricos y agregados
Usuarios	Muchos usuarios simultáneos	Pocos analistas
Tipo de consultas	Simples y frecuentes	Complejas y esporádicas
Actualización	Lectura/escritura constante	Principalmente lectura
Estructura	Normalizada (evita redundancia)	Desnormalizada (optimizada para consultas)

1.1.4.2 Ejemplos Prácticos

1.1.4.2.1 Tienda Online (E-commerce)

OLTP - Sistema Transaccional

Situación: Un cliente compra un producto



Operaciones OLTP:

1. Verificar stock disponible
2. Registrar el pedido
3. Actualizar inventario (stock - 1)
4. Procesar pago
5. Generar orden de envío

Características:

- Operación en tiempo real
- Afecta 1 registro específico
- Debe ser precisa e inmediata
- Si hay error, hacer rollback

OLAP - Sistema Analítico

Situación: El gerente quiere analizar ventas

Consulta OLAP:

"¿Cuáles son las categorías de productos más vendidas en los últimos 6 meses comparando regiones?"

Proceso:

- Analiza millones de transacciones históricas
- Agrupa por: categoría, mes, región
- Calcula totales, promedios, tendencias
- Genera gráficos comparativos

Resultado:

- Descubre que "Electrónica" vende 40% más en región Sur
- Identifica que ventas bajan 15% en agosto
- Ayuda a decidir: invertir más inventario en Sur

1.1.4.2.2 Cajero Automático (Banco)

Ejemplo 2: Cajero Automático (Banco)

OLTP

Situación: Dos personas con cuenta conjunta van al cajero

Persona A → Cajero 1: Quiere retirar \$1,000

Persona B → Cajero 2: Quiere retirar \$1,000

Saldo disponible: \$1,000

Sistema OLTP garantiza:

1. Solo UNA transacción se procesa
2. La más rápida gana
3. La segunda es rechazada (saldo insuficiente)
4. Integridad de datos mantenida

Esto requiere:

- Alta velocidad
- Control de concurrencia
- Transacciones ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad)

OLAP

```
Situación: El banco analiza tendencias
Consulta del analista:
"Comparar retiros en cajeros vs transferencias móviles
durante último año por grupo de edad"

Sistema OLAP:
- Procesa millones de transacciones históricas
- Cruza datos: tipo operación, edad, fecha, sucursal
- Descubre: Jóvenes prefieren app móvil 80%
- Conclusión: Invertir en mejoras de app, reducir cajeros
```

1.1.4.2.3 Sistema inventario (Retail)

OLTP

```
sql
-- Registro de venta inmediata
INSERT INTO ventas (producto_id, cantidad, fecha, cliente_id)
VALUES (890, 2, '2025-01-25', 567);

UPDATE inventario
SET stock = stock - 2
WHERE producto_id = 890;

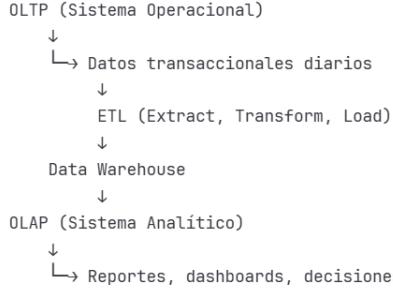
-- Operación: <100ms
-- Registros afectados: 1-2
```

OLAP

```
sql
-- Análisis de ventas multidimensional
SELECT
    categoria,
    region,
    MONTH(fecha) AS mes,
    SUM(cantidad) AS total_vendido,
    AVG(precio) AS precio_promedio
FROM
    ventas_historico
    JOIN productos USING(producto_id)
    JOIN clientes USING(cliente_id)
WHERE
    fecha BETWEEN '2024-01-01' AND '2024-12-31'
GROUP BY
    categoria, region, MONTH(fecha)
HAVING
    total_vendido > 1000
ORDER BY
    total_vendido DESC;

-- Operación: varios segundos
-- Registros analizados: millones
-- Resultado: insights para estrategia comercial
```

1.1.4.2.4 Relacion entre OLTP y OLAP



Flujo típico:

1. OLTP captura transacciones en tiempo real
2. ETL extrae datos de múltiples sistemas OLTP
3. Datos se limpian, transforman y cargan en Data Warehouse
4. OLAP permite análisis multidimensional
5. Gerentes toman decisiones basadas en insights

1.2 ¿Qué es un DWH?

Un Data Warehouse es una base de datos especial optimizada para análisis y generación de reportes, no para operaciones diarias.

1.2.1 ¿Por Qué Necesitamos un Data Warehouse?

Porque usamos los datos con dos propósitos diferentes:

1. Propósito Operativo (OLTP)

Hacer que el negocio funcione día a día:

- Recibir y procesar pedidos
- Atender reclamaciones
- Reponer inventario
- Mantener la rueda girando

2. Propósito Analítico (OLAP)

Entender y mejorar el negocio:

- ¿Cuál es nuestra mejor categoría de productos?
- ¿Cómo van las ventas este mes vs el mes pasado?
- ¿Qué podemos mejorar?
- Evaluar rendimiento y tomar decisiones futuras

Mientras OLTP hace girar la rueda, OLAP observa cómo gira y cómo mejorarlala

1.2.2 El Problema Sin Data Warehouse

Frases típicas en empresas sin DW:

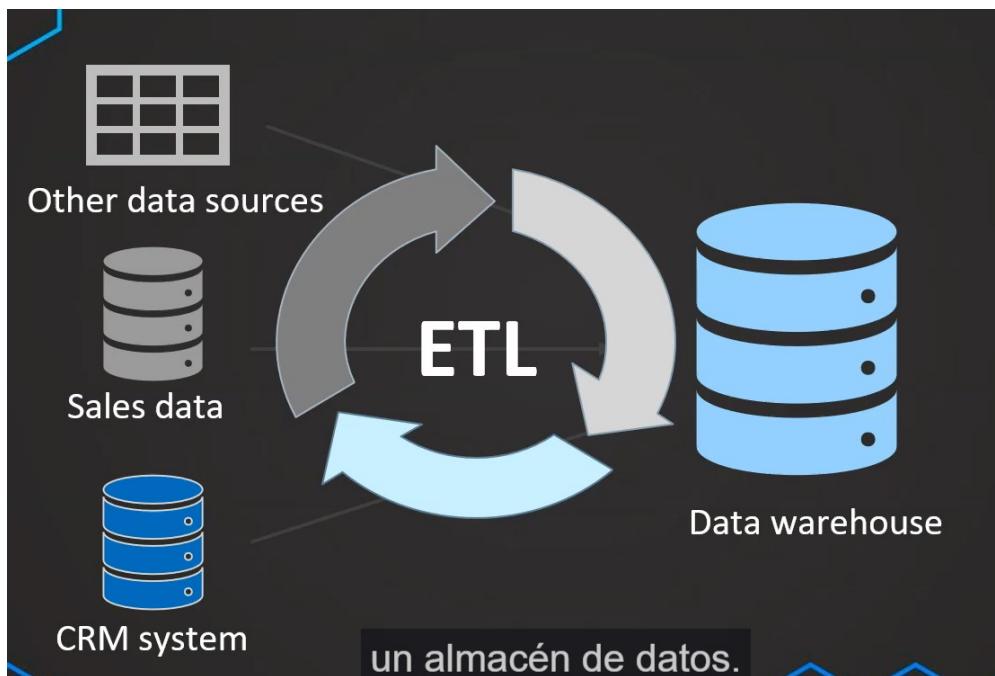
- ✗ "Tenemos muchos datos, pero no podemos usarlos"
- ✗ "Es muy complicado acceder a ellos"
- ✗ "Están repartidos en diferentes sistemas"
- ✗ "Solo quiero ver lo relevante de forma rápida"
- ✗ "Necesitamos decidir con hechos, no discutir sobre los números"

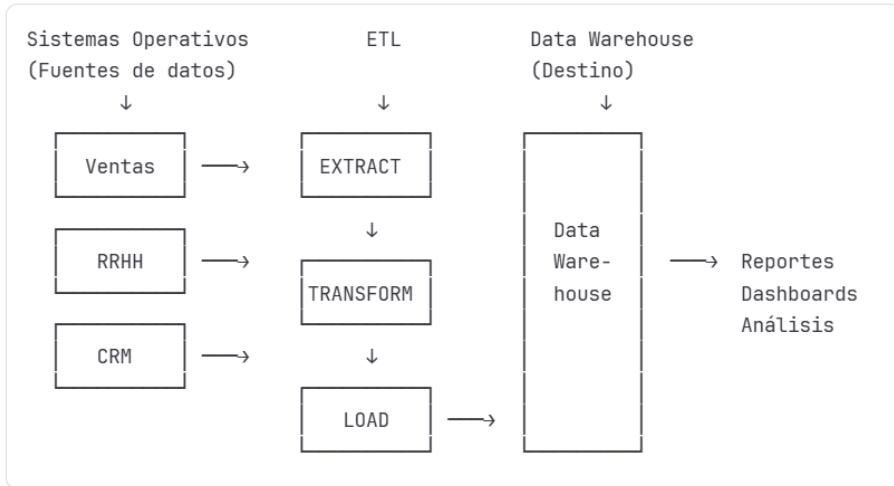
1.2.3 Características de un Data Warehouse

Característica	Descripción	Por qué importa
Fácil de usar	No debe ser super técnico	Analistas sin conocimientos profundos pueden trabajar
Nombres claros	Todo debe ser entendible	No confusión sobre qué significa cada dato
Consultas rápidas	Rendimiento optimizado	Analizar millones de registros en segundos
Datos centralizados	Todo en un solo lugar	No buscar en 10 sistemas diferentes
Optimizado para análisis	Estructura pensada para reportes	No para transacciones

1.2.4 El Proceso ETL (Extract, Transform, Load)

Este proceso consume el 80-90% del tiempo al crear un Data Warehouse.





1.2.4.1 E – Extract (Extraer)

Sacar datos de diferentes sistemas sin afectar su rendimiento.

Ejemplo práctico:

Tienes 3 sistemas:

- Sistema de Ventas (SQL Server)
- Sistema de RRHH (Oracle)
- Sistema CRM (Salesforce)

Extract copia los datos de estos 3 sistemas sin ralentizarlos.

1.2.4.2 T - Transform (Transformar)

Integrar y limpiar datos para que todos tengan la misma estructura.

Antes (datos sucios):

- Ventas: Cliente "Juan Pérez", Fecha "25/01/2025"
- CRM: Cliente "juan perez", Fecha "2025-01-25"
- RRHH: Cliente "JUAN PEREZ", Fecha "25-Ene-2025"

Después (datos limpios y estandarizados):

- Todos: Cliente "Juan Pérez", Fecha "2025-01-25"

Otras transformaciones:

- Eliminar duplicados
- Agregar datos (sumar ventas diarias → total mensual)
- Calcular métricas (total_venta = cantidad × precio)
- Validar calidad (rechazar registros con datos vacíos)

1.2.4.3 L – Load (Cargar)

Insertar los datos ya transformados en el Data Warehouse.

Cargar datos de ventas del día en el DW:

- Puede ser carga completa (todos los datos)
- O carga incremental (solo lo nuevo desde última carga)

Típicamente se hace:

- Diariamente (cada noche)
- Semanalmente
- Mensualmente

Según necesidad del negocio

1.2.5 Ejemplo Completo: Tienda de Retail

Sin Data Warehouse:

Gerente: "¿Cuánto vendimos en el mes?"

Analista:

1. Descarga datos de Sistema Ventas → Excel
2. Descarga datos de CRM → Excel
3. Descarga datos de Inventario → Excel
4. Limpia manualmente (3 horas)
5. Hace cruce de datos (2 horas)
6. Crea reporte (1 hora)

Total: 6 horas, datos posiblemente desactualizados o con errores

Con Data Warehouse:

Gerente: "¿Cuánto vendimos en el mes?" 

Analista:

```
SELECT
    mes,
    categoria,
    SUM(total_ventas) as ventas_totales
FROM
    ventas_mensuales
WHERE
    año = 2025
GROUP BY
    mes, categoria;
```

Total: 5 segundos, datos confiables y actualizados

1.2.6 Objetivos del Data Warehouse

Ubicación centralizada

- Todos los datos en un solo lugar
- Acceso coherente desde múltiples fuentes

Acceso rápido

- Consultas optimizadas
- Recuperación veloz de resultados

Fácil de usar

- Modelado simple y comprensible
- Nombres de tablas/campos claros

Proceso consistente y repetido

- ETL automatizado y programado
- Ejecución regular (ej: cada noche)

Generar valor

- Crear reportes
- Dashboards visuales
- Análisis para decisiones estratégicas

Goals of a data warehouse

- ✓ Centralized and consistent location for data
- ✓ Data must be accessible fast (query performance)
- ✓ User-friendly (easy to understand)
- ✓ Must load data consistently and repeatedly (ETL)
- ✓ Reporting and data visualization built on top

1.2.7 Flujo Típico en una Empresa

Lunes - Viernes (Operación)



Sistema OLTP captura transacciones diarias



Cada noche a las 2:00 AM



ETL extrae, transforma y carga en DW



Martes 8:00 AM



Analistas consultan DW actualizado



Generan reportes para gerencia



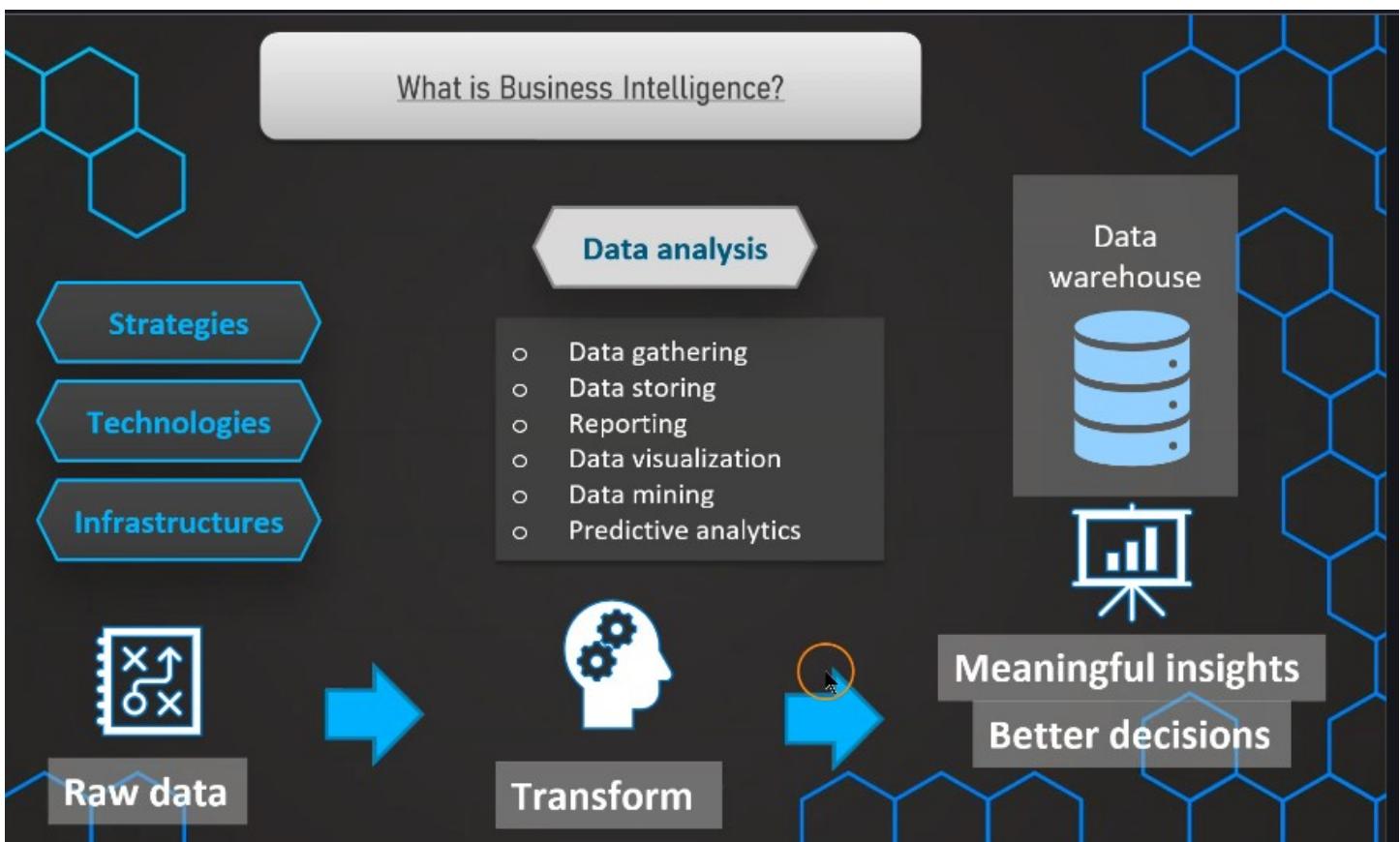
Gerencia toma decisiones basadas en datos

1.2.8 Conclusion

Un Data Warehouse es esencial para cualquier empresa que quiera tomar decisiones basadas en datos. El proceso ETL es el corazón que alimenta el DW con información limpia, consistente y lista para análisis.

Recordar: Si no puedes medir, no puedes mejorar. El DW te permite medir correctamente.

1.3 Business Intelligence (BI) y DWH vs Data Lake

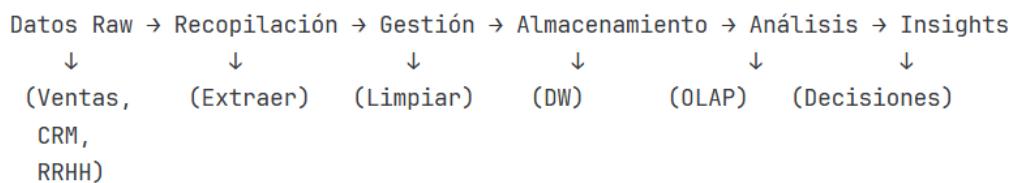


1.3.1 ¿Qué es Business Intelligence?

Business Intelligence es el conjunto de estrategias, tecnologías, procedimientos e infraestructuras que transforman datos en bruto en información útil para tomar mejores decisiones empresariales.

En pocas palabras: BI convierte montones de datos sin sentido en insights accionables.

1.3.2 Componentes de Business Intelligence



- **Recopilación:** Obtener datos de múltiples fuentes
- **Gestión:** Limpiar y organizar
- **Almacenamiento:** Guardar en Data Warehouse
- **Análisis:** Procesar con herramientas OLAP/Data Mining
- **Visualización:** Crear dashboards y reportes
- **Insights:** Tomar decisiones basadas en datos

1.3.3 Herramientas y Técnicas de BI + Ejemplo práctico

Componente	Descripción	Ejemplo
Visualización de datos	Gráficos, dashboards interactivos	Power BI, Tableau
Reportes	Informes automatizados	Crystal Reports
Data Mining	Descubrir patrones ocultos	Algoritmos de clustering
OLAP	Análisis multidimensional	Cubos OLAP
Análisis predictivo	Predecir comportamientos futuros	Machine Learning
KPIs	Indicadores clave de rendimiento	Ventas, conversión, ROI

Ejemplo práctico de BI

Sin BI (Empresa tradicional):

Gerente: "¿Cómo van las ventas?"



Proceso manual:

1. Pedir datos a ventas → 2 días
2. Pedir datos a finanzas → 1 día
3. Pedir datos a marketing → 1 día
4. Analista hace Excel → 3 días
5. Presenta reporte → Ya pasó 1 semana

Problemas:

- ✗ Datos desactualizados
- ✗ Errores manuales
- ✗ Versiones contradictorias
- ✗ Decisiones tardías

Con BI (Empresa moderna):

Gerente: Abre dashboard en Power BI



Ve en tiempo real:

- Ventas totales: \$2.5M ($\uparrow 15\%$ vs mes pasado)
- Producto top: iPhone 15 (40% del total)
- Región fuerte: Norte ($\uparrow 25\%$)
- Región débil: Sur ($\downarrow 10\%$)
- Tendencia: Viernes son el mejor día

Tiempo: 5 segundos

Decisión inmediata: "Invertir más inventario en Norte"

1.3.4 Ejemplo de BI en Empresas Reales

Netflix

BI en acción:

- Analiza qué ves, cuándo pausas, qué abandonas
- Predice qué series te gustarán
- Decide qué contenido producir

Resultado:

- 80% de contenido visto viene de recomendaciones
- Ahorran millones produciendo contenido que saben que funcionará

Amazon

BI en acción:

- "Quienes compraron esto también compraron..."
- Precios dinámicos según demanda
- Predicción de inventario por región

Resultado:

- 35% de ventas vienen de recomendaciones
- Reducción de costos de almacenamiento

Coca-Cola

BI en acción:

- Análisis demográfico de consumidores
- Optimización de rutas de distribución
- Lanzamiento de Coca-Cola Zero basado en datos

Resultado:

- Marketing personalizado por región
- Productos exitosos basados en datos reales

1.3.5 DWH en BI

El Data Warehouse es el componente MÁS IMPORTANTE de Business Intelligence.

¿Por qué?

Porque el DW es:

- Ubicación centralizada de datos estructurados
- Datos transformados listos para análisis
- Optimizado para consultas rápidas (OLAP)
- Base para visualizaciones y reportes



1.3.6 Data Lake vs DWH + Comparación Detallada

Ahora viene la diferencia que genera confusión.

Analogía Simple:

Data Warehouse = Supermercado organizado

- Todo en estantes ordenados
- Productos clasificados por categoría
- Fácil encontrar lo que buscas
- Solo productos útiles

Data Lake = Almacén gigante

- Todo amontonado como llegó
- Sin clasificar
- Tienes que buscar más
- Incluye de todo (útil y basura)

Comparación Detallada

Aspecto	Data Warehouse	Data Lake
Datos	Estructurados (tablas)	Todos los tipos (raw)
Procesamiento	ETL transforma ANTES de llegar al DW	ELT carga raw y transforma DENTRO
Esquema	Schema-on-write (definido)	Schema-on-read (flexible)
Usuarios	Analistas de negocio	Científicos de datos
Velocidad consulta	Muy rápida	Variable
Costo almacenamiento	Más caro (solo lo necesario)	Más barato (todo)
Propósito	Reportes, BI, análisis específico	Exploración, ML, Big Data
Ejemplo	SQL Server, Snowflake	Hadoop, AWS S3, Azure Data Lake
Analogía	Biblioteca organizada	Océano de datos

Aclaración Importante: ¿Dónde se Transforma?

Flujo ETL (Data Warehouse):

Sistemas Fuente → [ETL Tool limpia aquí] → Data Warehouse
(Talend, SSIS) (datos YA limpios)

El DW NO transforma, recibe datos ya procesados.

Flujo ELT (Data Lake):

Sistemas Fuente → Data Lake → [Transforma aquí dentro]
(datos raw) (Spark, SQL en el Lake)

El Lake Sí transforma, **tiene el poder para hacerlo.**

1.3.7 Ejemplos Prácticos Completos

1.3.7.1: Ejemplo 1: E-commerce - Data Warehouse:

Fuentes originales:

- Sistema POS: ventas diarias
- CRM: datos clientes
- Inventario: stock productos



ETL (Proceso):

1. Extraer ventas + clientes + stock
2. Transformar: limpiar, unificar fechas, calcular totales
3. Cargar en DW con estructura:

Tabla: ventas_mensuales

Mes	Producto	Ventas	Cliente
Ene-25	iPhone	\$100,000	500
Ene-25	Samsung	\$80,000	300

Consulta rápida:

```
SELECT mes, SUM(ventas)
FROM ventas_mensuales
WHERE año = 2025;
```

Resultado en 2 segundos

Para: Gerente de ventas (Power BI dashboard)

1.3.7.2: Ejemplo 1: E-commerce – Data Lake:

Datos almacenados (todo raw):

- Logs de servidor web (10GB/día)
- Clicks de usuarios (JSON)
- Imágenes de productos
- Videos de TikTok/Instagram
- Reviews en redes sociales (texto sin estructura)
- Datos de sensores IoT



Todo guardado tal cual llega, SIN transformar

Para: Científico de datos

Uso: Machine Learning para predecir churn

Análisis de sentimiento en reviews

Clustering de comportamiento usuarios

1.3.7.3: Ejemplo 2: Hospital - DWH:

Datos limpios y estructurados:

- Tabla: pacientes (nombre, edad, ID)
- Tabla: consultas (fecha, doctor, diagnóstico)
- Tabla: medicamentos (prescripción, dosis)



Reportes BI:

- ¿Cuántos pacientes con diabetes?
- ¿Tiempo promedio de espera?
- ¿Costo promedio por tratamiento?

Usuario: Director del hospital

Herramienta: Tableau

Consultas: Rápidas y precisas

1.3.7.4: Ejemplo 2: Hospital – Data Lake:



Datos diversos sin procesar:

- Imágenes de rayos X (GB de archivos)
- Scans de resonancia magnética
- Notas de médicos (texto libre)
- Audio de consultas
- Datos de dispositivos wearables
- Genoma de pacientes (archivos masivos)

Usuario: Investigadores médicos

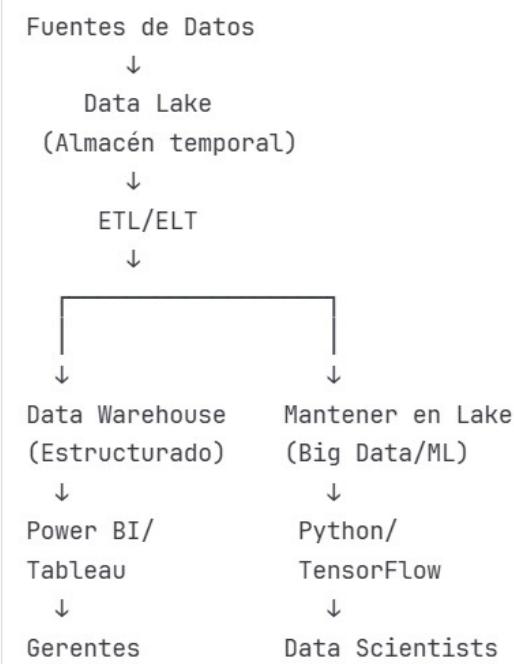
Uso: Entrenar IA para detectar cáncer en rayos X

Análisis genético para medicina personalizada

1.3.8 ¿Se Pueden Usar Juntos?

¡SÍ! De hecho, es lo IDEAL.

Arquitectura Moderna Híbrida:



Ejemplo: Empresa Tech

Data Lake (S3):

- Logs de app: 500GB/día
- Videos de usuarios: 1TB/día
- Datos de sensores: 200GB/día

↓

Algunos datos procesados → Data Warehouse:

- Métricas de uso diario
- KPIs de performance
- Costos operacionales

↓

BI Dashboard para CEO

Otros datos quedan en Lake:

- Videos para entrenar ML
- Logs para análisis de anomalías

↓

Científicos de datos

1.3.9 ¿Cuándo usar cada uno? + Flujo Completo de BI

Usa Data Warehouse cuando:

- Necesitas reportes rápidos y confiables
- Datos son principalmente estructurados
- Usuarios son analistas de negocio
- Necesitas dashboards en Power BI/Tableau
- Preguntas de negocio están claras

Ejemplo: "Ventas mensuales por región"

Usa Data Lake cuando:

- Tienes Big Data (terabytes/petabytes)
- Datos no estructurados (imágenes, videos, logs)
- No sabes qué harás con todos los datos
- Necesitas flexibilidad para experimentar
- Proyectos de Machine Learning/IA

Ejemplo: "Entrenar IA para reconocer fraude en imágenes"

Usa Ambos cuando:

- Empresa grande con múltiples necesidades
- Necesitas BI + Data Science
- Quieres lo mejor de ambos mundos

Ejemplo: Netflix, Amazon, Google

Flujo Completo de BI

1. DATOS RAW (múltiples fuentes)
↓
2. DATA LAKE (almacena TODO)
↓
3. ETL (procesa lo importante)
↓
4. DATA WAREHOUSE (datos limpios)
↓
5. OLAP/BI Tools (análisis)
↓
6. DASHBOARDS (visualización)
↓
7. DECISIONES (acción)



1.3.10 Resumen en 3 Puntos + Conclusión

1. **Business Intelligence** = Proceso completo de convertir datos en decisiones
2. **Data Warehouse** = Componente principal de BI

- Datos limpios y estructurados
- Para reportes rápidos
- Usuarios: analistas de negocio

3. Data Lake = Complemento para Big Data

- Datos raw sin procesar
- Para exploración y ML
- Usuarios: científicos de datos

Lo ideal: Usar ambos en una arquitectura híbrida moderna.

Conclusión

- BI sin Data Warehouse = Imposible hacer análisis eficientes
- Data Warehouse sin BI = Datos guardados pero no usados
- Data Lake = Complemento poderoso para casos avanzados
- Juntos = Potencia máxima para decisiones basadas en datos

Recordar: El objetivo final de todo esto es tomar mejores decisiones más rápido basándose en datos confiables, no en intuición.

1.3.11 + Propiedades y Condiciones de los Datos: Data Warehouse vs Data Lake

Las 7 Propiedades Clave que Determinan el Destino

1.3.11.1 Estructura de los datos

Data Warehouse: SOLO Estructurados

Condición: Datos que se pueden organizar en tablas/filas/columnas

Datos Estructurados:

- Bases de datos relacionales (SQL)
- Hojas de cálculo (Excel, CSV)
- Transacciones (compras, pagos)
- Formularios (nombre, edad, dirección)
- ERPs, CRMs (datos transaccionales)

Ejemplo concreto:

Cliente_ID	Producto	Cantidad	Precio
1001	iPhone	2	\$2000
1002	MacBook	1	\$1500

¿Por qué al DWH?

- Se puede predefinir el esquema
- Fácil hacer SUM, AVG, COUNT
- Consultas SQL rápidas

Data Lake: TODOS los Tipos

Condición: Cualquier dato, sin importar su formato



Estructurados: Tablas, CSV

Semi-estructurados: JSON, XML, logs

No estructurados: Imágenes, videos, audio, texto libre

Ejemplos concretos:

Semi-estructurado (JSON):

```
{  
  "usuario": "juan123",  
  "clicks": [  
    {"pagina": "inicio", "tiempo": 3.5},  
    {"pagina": "productos", "tiempo": 12.3}  
  ]  
}
```

No estructurado:

- Foto de producto (5MB .jpg)
- Video de TikTok (50MB .mp4)
- Audio de call center (10MB .wav)
- PDF de contrato (2MB)
- Tweet: "Me encanta este producto! 😊"

¿Por qué al Data Lake?

- No necesitas estructurarlo primero
- Guardas "tal cual" llegó
- Decides después qué hacer con ello

1.3.11.2 Propósito / Finalidad

Data Warehouse: Propósito DEFINIDO

Condición: Sabes PARA QUÉ usarás los datos

Preguntas claras que responderá:

- ¿Cuánto vendimos este mes?
- ¿Cuál es el cliente que más compra?
- ¿Qué producto tiene mejor margen?
- ¿Cuál es la tasa de retención?

Característica: INTENCIÓN CLARA

Ejemplo:

Empresa retail sabe que necesita:

- Reporte de ventas mensuales
- Dashboard de inventario
- Análisis de clientes top 10

Entonces diseña tablas específicas:

- ventas_mensuales
- inventario_actual
- clientes_top

→ Al Data Warehouse

Data Lake: Propósito INDEFINIDO o EXPLORATORIO

Condición: NO sabes qué preguntarás o quieres EXPLORAR

Situaciones:

- ? "Guardemos todos los logs, tal vez sean útiles"
- ? "No sé qué insight hay aquí, pero lo guardamos"
- ? "Quiero experimentar con estos datos"
- ? "Tal vez descubramos patrones ocultos"

Característica: DESCUBRIMIENTO

Ejemplo:

Startup tech guarda:



- TODOS los logs de servidor (GB/día)
- TODOS los clicks de usuarios
- TODAS las búsquedas que hacen

¿Para qué? Aún no lo saben bien

Futuro: Entrenar IA, detectar patrones, experimentos

→ Al Data Lake

1.3.11.3 Esquema (schema)

Data Warehouse: Schema-on-WRITE

Condición: Defines la estructura ANTES de guardar

Proceso:

1. Diseñas tablas (columnas, tipos de datos)
2. Defines relaciones (claves foráneas)
3. Estableces restricciones (NOT NULL, UNIQUE)
4. LUEGO cargas los datos

Si datos no cumplen esquema → RECHAZO

Ejemplo:

```
CREATE TABLE ventas (
    venta_id INT PRIMARY KEY,
    fecha DATE NOT NULL,
    monto DECIMAL(10,2),
    cliente_id INT FOREIGN KEY
);
-- Si intentas cargar texto en "monto" → ERROR
```

Ventaja: Datos siempre consistentes y confiables

Desventaja: Rígido, difícil cambiar después

Data Lake: Schema-on-READ

Condición: Guardas primero, defines estructura DESPUÉS



Proceso:

1. Recibes datos → Guardas tal cual
2. Más tarde, cuando necesites analizarlos
3. ENTONCES defines cómo interpretarlos

Ejemplo:

```
-- Guardas archivo JSON raw:  
archivo_usuario_123.json
```

-- Cuando lo lees, decides:

¿Es un usuario?

¿Es una transacción?

¿Tiene clicks?

→ TÚ decides en tiempo de lectura

Ventaja: Ultra flexible, adaptable

Desventaja: Puede ser caótico (Data Swamp)

1.3.11.4 Calidad y Limpieza de datos

Data Warehouse: Alta CALIDAD Garantizada

Condición: Datos LIMPIOS y VALIDADOS antes de entrar



Proceso ETL previo:

- Elimina duplicados
- Valida formatos (fechas, emails)
- Normaliza valores (MAYÚSCULAS → Título)
- Rechaza datos erróneos
- Completa datos faltantes

Ejemplo:

Datos sucios:

- Cliente: "juan perez", "JUAN PEREZ", "J. Pérez"
- Fecha: "25/01/2025", "2025-01-25", "25-Ene-2025"

Datos en DWH (después de ETL):

- Cliente: "Juan Pérez"
- Fecha: "2025-01-25"

Resultado: 100% confiable para reportes

Data Lake: Calidad VARIABLE (As-Is)

Condición: Datos en su estado ORIGINAL, sin limpiar



Se guarda TODO:

- Con errores
- Con duplicados
- Incompleto
- En formatos inconsistentes

Ejemplo:

Log de servidor (raw):

```
192.168.1.1 - - [25/Jan/2025:10:30:45] "GET /api/user" 200
ERROR: Connection timeout
192.168.1.2 - - [25/Jan/2025:10:30:46] "POST /api/order" 500
Warning: Slow query detected
```

→ Guardas tal cual, basura incluida

Limpieza: Más tarde, si lo necesitas

1.3.11.5 Usuarios y Habilidades

Data Warehouse: Usuarios de NEGOCIO

Condición: Diseñado para NO-técnicos



Usuarios típicos:

- Gerentes de ventas
- Analistas de negocio
- Directores financieros
- Gerentes de marketing
- Cualquier tomador de decisiones

Habilidades requeridas:

- Saber usar Power BI / Tableau (clicks)
- SQL básico (opcional)
- Entender gráficos y KPIs

Ejemplo de uso:

Gerente abre dashboard → Ve ventas → Listo

NO necesita:

- Programar en Python
- Conocer algoritmos ML
- Limpiar datos

Data Lake: Usuarios TÉCNICOS

Condición: Requiere expertise técnico alto

Usuarios típicos:

- Científicos de datos
- Ingenieros de ML
- Investigadores
- Data Engineers

Habilidades requeridas:

- Python / R / Scala
- Spark, Hadoop
- Machine Learning
- Estadística avanzada
- Limpieza de datos complejos

Ejemplo de uso:

Data Scientist:

1. Busca datos en el Lake
2. Los limpia con Python
3. Entrena modelo ML
4. Experimenta y descubre insights

Necesita:

- Código (PySpark, pandas)
- Paciencia (datos sucios)
- Creatividad (exploración)

1.3.11.6 Velocidad de cambio / Agilidad

Data Warehouse: RÍGIDO y ESTABLE

Condición: Cambiar el esquema es COSTOSO y LENTO



Escenario de cambio:

"Necesito agregar nueva columna: email_secundario"

Proceso:

1. Modificar esquema de tabla (DDL)
2. Actualizar proceso ETL
3. Retrocesar datos históricos
4. Validar integridad referencial
5. Actualizar reportes/dashboards
6. Probar todo de nuevo

Tiempo: Semanas o meses

Costo: Alto (recursos IT)

¿Por qué es difícil?

- Datos ya estructurados
- Relaciones entre tablas
- Reportes dependen de estructura actual

Data Lake: ÁGIL y FLEXIBLE

Condición: Cambiar es RÁPIDO y BARATO



Escenario de cambio:

"Necesito analizar un nuevo tipo de dato"

Proceso:

1. Carga el nuevo archivo/stream
2. Listo

Tiempo: Minutos u horas

Costo: Mínimo

Ejemplo:

Día 1: Guardas logs de servidor
Día 10: Decides guardar también videos
Día 20: Ahora también archivos de audio

- Solo agregas más archivos
- NO rompes nada existente
- Máxima agilidad

1.3.11.7 Casos de uso / Aplicaciones

Data Warehouse: BI, Reporting, Operacional

Condición: Respuestas a preguntas CONOCIDAS y REPETITIVAS

Casos de uso:

- ─ Reportes mensuales de ventas
- ─ Dashboards ejecutivos (KPIs)
- ─ Análisis de tendencias históricas
- ─ Reportes regulatorios (compliance)
- ─ Forecasting basado en históricos
- ─ Segmentación de clientes

Tipo de análisis: DESCRIPTIVO

- "¿Qué pasó?"
- "¿Cuánto vendimos?"
- "¿Cuál es la tendencia?"

Herramientas:

- Power BI
- Tableau
- Qlik
- Looker

Ejemplo real:

Cadena de supermercados:

- Reporte diario de ventas por tienda
- Dashboard de productos con bajo stock
- Análisis mensual de margen de ganancia
- Comparación ventas año actual vs anterior

Todos los días, mismos reportes

- Data Warehouse perfecto

Data Lake: ML, IA, Research, Exploración

Condición: Preguntas ABIERTAS o FUTURAS, experimentación

Casos de uso:

- 🤖 Entrenar modelos de Machine Learning
- 🤖 Análisis de sentimiento (redes sociales)
- 🤖 Reconocimiento de imágenes
- 🤖 Procesamiento de lenguaje natural (NLP)
- 🤖 Detección de anomalías / fraude
- 🤖 Investigación científica
- 🤖 IoT y análisis de sensores

Tipo de análisis: PREDICTIVO / PRESCRIPTIVO

"¿Qué pasará?"
"¿Por qué pasó?"
"¿Qué debemos hacer?"

Herramientas:

- Python / R
- TensorFlow / PyTorch
- Spark MLlib
- Jupyter Notebooks

Ejemplo real:

Netflix:



- Analiza TODOS los videos que ves
- Cuándo pausas, adelantas, retrocedes
- Qué abandonas a los 5 minutos

Usan Data Lake para:

- Entrenar algoritmo de recomendación
- Predecir qué serie te gustará
- Decidir qué contenido producir

Preguntas cambian constantemente

- Data Lake perfecto

1.3.11.7.1 Análisis Descriptivo vs Prescriptivo + Tabla Comparativa

1.3.11.7.1.1 Análisis Descriptivo

Es el nivel más básico y común. Su objetivo es resumir datos históricos para entender eventos que ya ocurrieron. No intenta explicar por qué sucedieron las cosas ni qué pasará después; simplemente presenta los hechos de forma digerible.

- **Herramientas comunes:** Medidas de tendencia central (media, mediana), desviación estándar, histogramas y dashboards de Business Intelligence (BI).
- **Ejemplo:** Un informe que muestra que las ventas bajaron un 10% el mes pasado o un gráfico que indica que el uso de CPU en un servidor tuvo picos a las 3:00 AM.

1.3.11.7.1.2 Análisis Prescriptivo

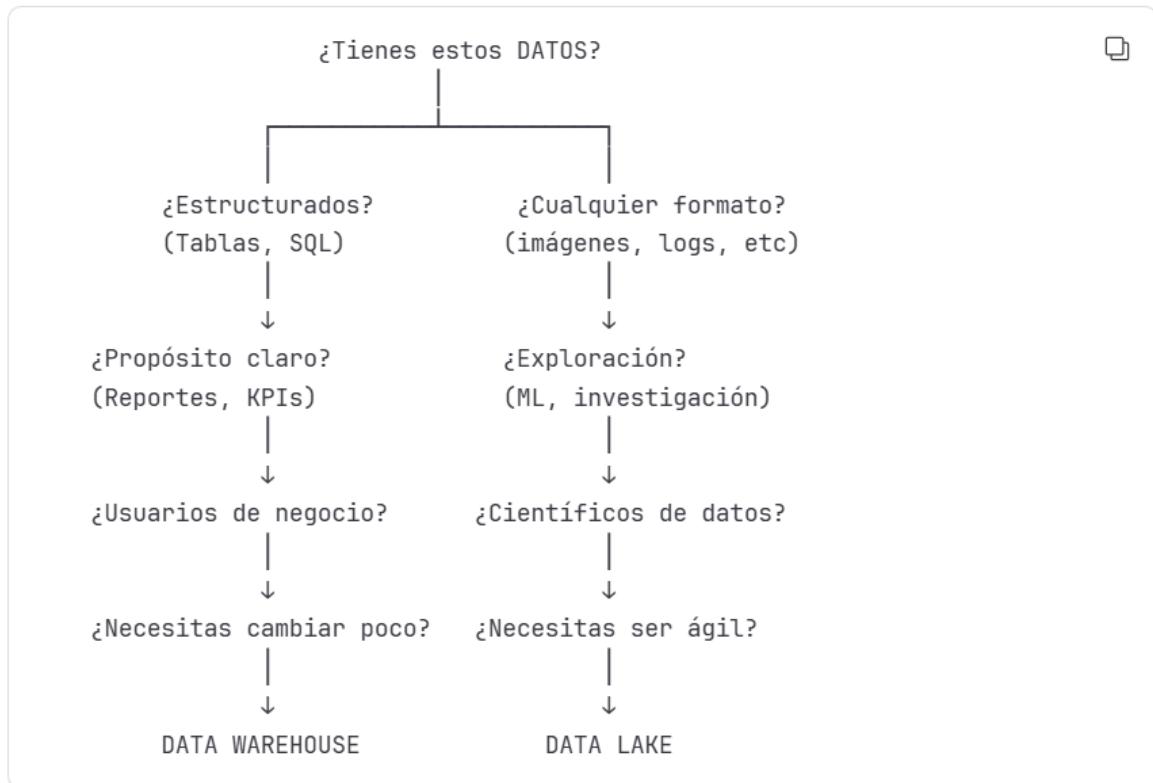
Este es el nivel más avanzado. No solo predice el futuro (como lo haría el análisis *predictivo*), sino que **sugiere acciones específicas** para obtener un resultado óptimo. Utiliza algoritmos complejos para evaluar múltiples escenarios y recomendar la mejor ruta a seguir.

- **Herramientas comunes:** Investigación de operaciones, motores de recomendación, redes neuronales, simulación de Monte Carlo y algoritmos de optimización.
- **Ejemplo:** Un sistema logístico que no solo sabe que habrá tráfico (predicción), sino que recalcula automáticamente la ruta de entrega para ahorrar combustible y tiempo (prescripción).

1.3.11.7.1.3 Comparativa Rapida

Característica	Análisis Descriptivo	Análisis Prescriptivo
Pregunta clave	¿Qué pasó?	¿Qué debemos hacer para que pase X?
Enfoque	Pasado / Presente	Futuro / Acción
Complejidad	Baja	Muy Alta
Valor agregado	Visibilidad y contexto	Ventaja competitiva y decisiones

1.3.11.8 RESUMEN: Decision Tree



1.3.11.9 Ejemplos Integrales por Industria

1.3.11.9.1 Retail / E-commerce

Data Warehouse:

Entidades:

- Clientes (ID, nombre, email, segmento)
- Productos (SKU, nombre, categoría, precio)
- Ventas (fecha, cantidad, monto, descuento)
- Tiendas (ubicación, región, tipo)



Propiedades:

- Estructurado (tablas relacionales)
- Propósito: Reportes de ventas
- Usuarios: Gerentes de ventas
- Calidad: Datos limpios (sin duplicados)
- Cambios: Pocas veces al año

Consultas típicas:

- Top 10 productos más vendidos
- Ventas por región este trimestre
- Clientes que gastaron más de \$10K

Data Lake:

Entidades:

- Clicks en web (millones/día, JSON)
- Imágenes de productos (GB)
- Reviews de clientes (texto libre)
- Videos de productos (TB)
- Logs de búsquedas (sin estructura)
- Datos de sensores IoT en tiendas

Propiedades:

- No estructurado / semi-estructurado
- Propósito: Descubrir patrones, ML
- Usuarios: Data Scientists
- Calidad: Variable (raw)
- Cambios: Constantes (añadir más datos)

Proyectos típicos:

- Sistema de recomendación (ML)
- Análisis de sentimiento en reviews
- Detección de fraude en transacciones
- Optimización de precios dinámicos

1.3.11.9.2 Healthcare

Data Warehouse:

Entidades:

- Pacientes (nombre, edad, historial)
- Citas (fecha, doctor, diagnóstico)
- Medicamentos (prescripción, dosis, fecha)
- Facturación (costos, seguros, pagos)



Propiedades:

- Altamente estructurado (HIPAA compliance)
- Propósito: Reportes operacionales
- Usuarios: Administradores hospitalarios
- Calidad: Crítica (datos médicos)
- Cambios: Controlados y auditados

Consultas típicas:

- Tiempo promedio de espera
- Ocupación de camas por departamento
- Costos por tipo de tratamiento
- Pacientes readmitidos en 30 días

Data Lake:

Entidades:



- Imágenes médicas (Rayos X, MRI, CT scans)
- Genoma de pacientes (archivos gigantes)
- Notas de médicos (texto libre)
- Audio de consultas
- Datos de wearables (Apple Watch, Fitbit)
- Investigación clínica (papers, estudios)

Propiedades:

- Mayormente no estructurado
- Propósito: Investigación, ML médico
- Usuarios: Investigadores, data scientists
- Calidad: Variable
- Cambios: Experimentación constante

Proyectos típicos:

- IA para detectar cáncer en rayos X
- Medicina personalizada (análisis genómico)
- Predicción de readmisiones
- Descubrimiento de nuevos tratamientos

1.3.11.10 Reglas Finales para Decidir + Conclusión

¿Va al Data Warehouse?

Responde Sí a TODAS:

2. ¿Los datos son estructurados? (filas/columnas)
3. ¿Sabes para qué los usarás? (reporte claro)
4. ¿Los usuarios son analistas de negocio?
5. ¿Necesitas respuestas rápidas y confiables?
6. ¿El esquema cambiará poco?

→ **DATA WAREHOUSE**

¿Va al Data Lake?

Responde Sí a CUALQUIERA:

1. ¿Son datos no estructurados? (imágenes, videos, logs)
2. ¿No sabes qué harás con ellos aún?
3. ¿Los usarán científicos de datos / ML?
4. ¿Necesitas flexibilidad para experimentar?
5. ¿Son datos masivos de IoT/sensores/streaming?

→ **DATA LAKE**

Conclusión

La propiedad fundamental que determina el destino:

Data Warehouse = Datos que tienen VALOR CONOCIDO

Data Lake = Datos que tienen VALOR POR DESCUBRIR

No es solo sobre estructura. Es sobre:

- **Intención:** ¿Sabes qué preguntarás?
- **Usuarios:** ¿Quién los usará?
- **Agilidad:** ¿Necesitas cambiar rápido?
- **Calidad:** ¿Necesitas garantía o tolerancia?

Lo ideal: Usar ambos en conjunto, cada uno cumple su rol específico y complementario.

1.4 Data Lake vs DWH

	Data Lake	Data Warehouse
Data	Raw	Processed
Technologies	Big data	Database
Structure	Unstructured	Structured
Usage	Not defined yet	Specific & ready to be used
Users	Data Scientists	Business users & IT

⚠ Concepto Erróneo Común

Mucha gente piensa que:

- ✗ "Data Lake reemplaza al Data Warehouse"
- ✗ "Son lo mismo"
- ✗ "Solo necesito uno de los dos"

FALSO. Son tecnologías **complementarias**, no excluyentes.

1.4.1 Diferencias Principales

1. Tipos de Datos

1. Tipo de Datos

Data Lake:

Datos SIN PROCESAR (raw)

- Salen directamente de los sistemas
- NO se transforman antes de guardar
- Se almacenan "tal cual"

Ejemplo:

Sistema de ventas genera:

ventas_raw_20250125.json

→ Directo al Data Lake

→ SIN limpiar, SIN validar

Data Warehouse:

Datos PROCESADOS y LIMPIOS

- Pasan por ETL primero
- Se transforman ANTES de guardar
- Listos para usar

Ejemplo:

ventas_raw_20250125.json

→ ETL limpia, valida, estructura

→ Tabla: ventas_mensuales (SQL)

→ AL Data Warehouse

2. Estructura de Datos

Data Lake:

NO estructurados o SEMI-estructurados

Tipos de archivos:

- 📁 CSV (miles de columnas sin orden)
- 📁 JSON (jerarquías complejas)
- 📁 Imágenes (.jpg, .png)
- 📁 Videos (.mp4, .avi)
- 📁 Logs de servidor (texto libre)
- 📁 Audio (.wav, .mp3)
- 📁 Archivos XML

No están en tablas relacionales

Data Warehouse:

SÓLO estructurados (tablas)

Estructura clara:

Columna1	Columna2	Columna3	Columna4
Dato	Dato	Dato	Dato

Bases de datos relacionales (SQL)

Tablas con filas y columnas bien definidas

3. Propósito / Caso de Uso

Data Lake:

Caso de uso MENOS DEFINIDO o FUTURO

Mentalidad:

- "Guardemos esto por si acaso"
- "Tal vez sea útil después"
- "Diferentes personas tendrán diferentes ideas"
- "Experimentos futuros"

Ejemplo:

Startup guarda TODOS los logs:
¿Para qué? Aún no lo saben exactamente
Tal vez para ML, tal vez para auditoría,
tal vez para detectar fraudes... veremos

→ Data Lake

Data Warehouse:

Caso de uso MUY ESPECÍFICO y CLARO desde el inicio

Mentalidad:

- "Necesitamos reporte X"
- "Dashboard para gerente Y"
- "KPI Z cada semana"

Ejemplo:

Empresa retail necesita:

- Reporte de ventas mensuales
- Dashboard de inventario
- Análisis de clientes top

Objetivo claro desde el día 1

→ Data Warehouse

4. Calidad de Datos

Data Lake:

Calidad NO garantizada
Es más DIFÍCIL gestionar

Problemas típicos:
✗ Datos duplicados
✗ Formatos inconsistentes
✗ Valores faltantes
✗ Errores sin validar
✗ Difícil navegar
✗ "Data Swamp" (pantano de datos)

Riesgo: Convertirse en basura digital

Data Warehouse:

Calidad ALTA garantizada
Fácil de gestionar

Garantías:

- Sin duplicados
- Formatos consistentes
- Valores validados
- Estructura clara
- Fácil navegar
- Confiable para decisiones

5. Usuarios

Data Lake:

Usuarios: CIENTÍFICOS DE DATOS

Habilidades necesarias:

-  Python/R/Scala
-  Spark, Hadoop
-  Machine Learning
-  Limpieza de datos
-  Paciencia técnica

Perfil:

- Nivel de competencia: ALTO
- Cómodos con datos "sucios"
- Experimentación constante

Data Warehouse:

Usuarios: USUARIOS DE NEGOCIO

Habilidades necesarias:

-  Power BI / Tableau (clicks)
-  SQL básico (opcional)
-  Leer gráficos

Perfil:

- Nivel de competencia: MEDIO-BAJO
- Necesitan datos limpios
- Reportes predefinidos
- Alta facilidad de uso

6. Tecnologías

Data Lake:

Tecnologías de BIG DATA



Stack típico:

- Hadoop (almacenamiento distribuido)
- Spark (procesamiento)
- AWS S3 / Azure Data Lake
- Apache Parquet
- Delta Lake

Razón:

- Grandes volúmenes (TB/PB)
- Datos no estructurados
- Procesamiento paralelo masivo

Data Warehouse:

Bases de datos tradicionales (optimizadas)



Stack típico:

- SQL Server
- Oracle
- Snowflake
- Amazon Redshift
- Google BigQuery
- PostgreSQL

Razón:

- Alto rendimiento de consulta
- OLAP optimizado
- SQL estándar

1.4.2 Ejemplo Práctico Completo: E-commerce

Flujo Completo de Datos

DÍA 1: Operación Normal

Sistema Web genera:

- 10 millones de clicks (JSON)
- 1 millón de búsquedas (logs)
- 100k compras (transacciones)
- 50k imágenes subidas por usuarios
- 5k videos de productos

TODO va → Data Lake (AWS S3)

```
└── /raw(clicks/2025-01-25(clicks.json
└── /raw(searches/2025-01-25/search.log
└── /raw(purchases/2025-01-25/purchases.json
└── /raw(images/products/*.jpg
└── /raw/videos/reviews/*.mp4
```

Estado: Raw, sin procesar

Volumen: 500 GB en un día

DÍA 2: Proceso ETL (cada noche)

ETL toma del Data Lake:

1. Solo purchases.json (transacciones)
2. Limpia duplicados
3. Valida montos
4. Calcula totales
5. Estructura en tabla SQL

Resultado → Data Warehouse

Fecha	Producto	Ventas	Cliente
25-Ene-25	iPhone	\$500,000	2,500
25-Ene-25	MacBook	\$800,000	1,200

Estado: Limpio, estructurado

Volumen: 5 GB (solo lo importante)

DÍA 3: Uso de Datos

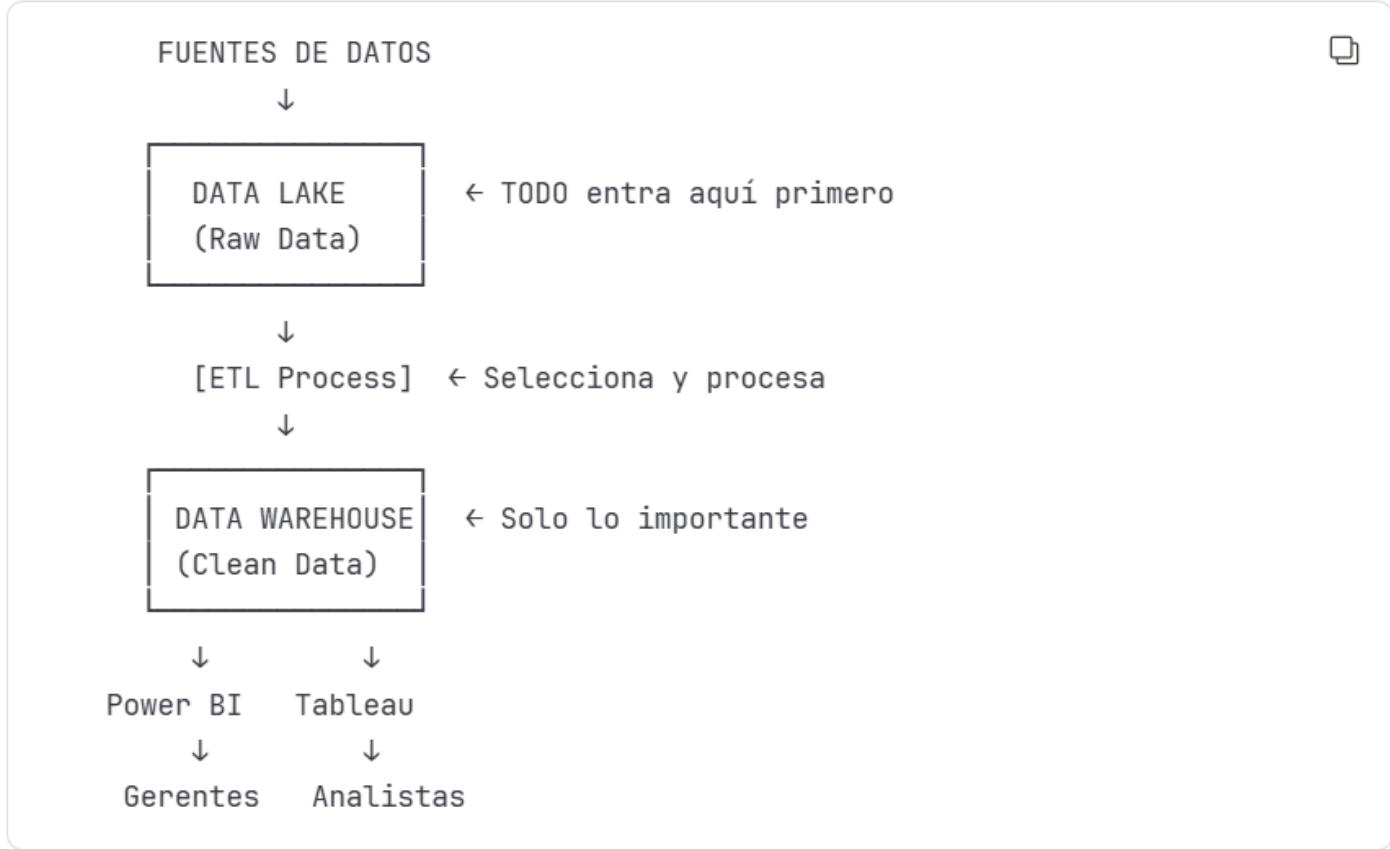
Data Warehouse:

- └ Gerente de Ventas
 - └ Abre Power BI
 - └ Dashboard: "Ventas de ayer"
 - └ 2 segundos ✓
- └ Director Financiero
 - └ Reporte mensual
 - └ SQL query rápida ✓

Data Lake:

- └ Data Scientist
 - └ Python + Spark
 - └ Entrena ML: Sistema de recomendación
 - └ Usa: clicks + searches + purchases
 - └ Procesa 500GB raw
 - └ 3 horas de procesamiento
 - └ Investigador de UX
 - └ Analiza videos de reviews
 - └ Procesamiento de lenguaje natural
 - └ Sentimiento de clientes

1.4.3 NO Son Mutuamente Excluyentes



Ventajas de usar AMBOS:

1. **Data Lake** = Máxima flexibilidad
 - o Guardas TODO por si acaso
 - o Experimentos de ML
 - o Análisis futuros desconocidos
2. **Data Warehouse** = Máxima eficiencia
 - o Reportes rápidos y confiables
 - o Usuarios de negocio felices
 - o Decisiones basadas en datos limpios

1.4.4 Desventajas del Data Lake (por qué necesitas DWH) – 3 Problemas

Sin Data Warehouse, solo Data Lake:

Problema 1: LENTO

Gerente: "¿Cuánto vendimos ayer?"

Sin DWH: Tienes que procesar 500GB raw cada vez

→ 30 minutos de espera ✗

Con DWH: Query SQL instantánea

→ 2 segundos ✓

Problema 2: CALIDAD

Analista: "Quiero top 10 productos"

Sin DWH: Datos con duplicados, errores

→ Resultados no confiables ✗

Con DWH: Datos validados y limpios

→ 100% confiable ✓



Problema 3: USUARIOS

CEO: "Muéstrame el dashboard"

Sin DWH: Necesitas un científico de datos

para extraer y limpiar cada vez ✗

Con DWH: CEO abre Power BI

→ Click y listo ✓

1.4.5 Reglas de Decisión

Usa SOLO Data Lake cuando:

- Startup pequeña
- Solo científicos de datos
- Principalmente ML/AI
- No necesitas reportes frecuentes
- Presupuesto limitado al inicio

Usa SOLO Data Warehouse cuando:

- Empresa tradicional
- Solo datos estructurados
- No tienes Big Data
- Principalmente BI/reporting
- Usuarios de negocio principalmente

Usa AMBOS cuando: (RECOMENDADO)

- Empresa mediana/grande
- Datos estructurados + no estructurados
- Necesitas BI + ML
- Múltiples tipos de usuarios
- Quieres lo mejor de ambos mundos

1.4.6 Comparación Lado a Lado

Característica	Data Lake	Data Warehouse
Datos	Raw, sin procesar	Procesados, limpios
Estructura	No estructurados	Estructurados (tablas)
Propósito	Indefinido/futuro	Específico/claro
Calidad	Variable, riesgosa	Alta, garantizada
Usuarios	Data Scientists	Analistas de negocio
Tecnología	Hadoop, Spark	SQL, OLAP
Velocidad consulta	Variable (lenta)	Muy rápida
Facilidad uso	Difícil (técnico)	Fácil (user-friendly)
Volumen	Masivo (PB)	Grande (TB)
Costo storage	Bajo (\$/GB)	Alto (\$/GB)
Casos de uso	ML, exploración	BI, reportes

1.4.7 Ejemplo Real: Netflix

Data Lake:

Guardan TODO:

- Videos completos (petabytes)
- Cada click de cada usuario (millones/segundo)
- Cuando pausas, adelantas, retrocedes
- Subtítulos en 40 idiomas
- Audio en múltiples versiones
- Metadatos de contenido
- Reviews y ratings
- Búsquedas fallidas

Uso:

- Algoritmo de recomendación (ML)
- Decisiones de contenido a producir
- Optimización de streaming
- A/B testing constante

Data Warehouse:



Extraen lo esencial:

- Usuarios activos por día
- Tiempo promedio de visualización
- Contenido más visto por región
- Tasa de cancelación
- Ingresos por región
- Costos de licencias

Uso:

- Dashboard ejecutivo para CEO
- Reportes financieros
- KPIs de negocio
- Decisiones estratégicas rápidas

Resultado:

- Data Lake: Innovación y ML
- Data Warehouse: Operación y reportes
- **Juntos:** Netflix domina el mercado

1.4.8 Conclusión

Data Lake ≠ Data Warehouse

Son **complementarios**, no **competidores**.

La mejor estrategia:

Data Lake: Guardas TODO

↓

(ETL)

↓

Data Warehouse: Extraes LO IMPORTANTE

↓

Power BI/Tableau: VISUALIZAS

↓

Decisiones: ACTÚAS

Mensaje clave del instructor:

"Un Data Lake NO reemplaza un Data Warehouse. Son diferentes tecnologías para diferentes propósitos.
Usa ambos para maximizar el valor de tus datos."

Recordar:

- Data Lake = Flexibilidad y experimentación
- Data Warehouse = Confiabilidad y velocidad
- Juntos = Éxito en BI

1.4.9 Pregunta 1: ¿big data se usa en contexto de data lake?

Si, principalmente.

Big Data se asocia MAS con Data Lake:

Big Data = 3 V's

- └ Volumen (TB/PB)
- └ Velocidad (streaming, real-time)
- └ Variedad (estructurado + NO estructurado)

Data Lake:

- Diseñado para Big Data
- Hadoop, Spark (tecnologías Big Data)
- Almacena TODO: imágenes, videos, logs
- Petabytes de datos

Data Warehouse:

- Puede manejar grandes volúmenes
- SOLO datos estructurados
- No todas las 3 V's

Excepción:

DWH modernos cloud (Snowflake, BigQuery):

- SÍ manejan volúmenes Big Data
- PERO solo datos estructurados
- NO todas las características de Big Data



Resumen:

- **Big Data completo** → Data Lake
- **Solo volumen grande** → DWH cloud también puede

1.4.10 Pregunta 2-Respuesta: Spark ES una herramienta de Big Data

Sí, Spark ES una herramienta de Big Data.

Spark = Motor de procesamiento distribuido para Big Data

Apache Spark:

- Diseñado específicamente para Big Data
- Procesa datos masivos en paralelo
- Distribuye trabajo en cientos/miles de nodos
- Maneja TB/PB de datos



Casos de uso:

- Procesar logs de millones de usuarios (Big Data)
- Entrenar modelos ML con GB/TB de datos
- Analizar streaming en tiempo real (Kafka + Spark)
- ETL de Data Lake (volúmenes masivos)

Stack típico Big Data:

Almacenamiento: Hadoop HDFS / Data Lake

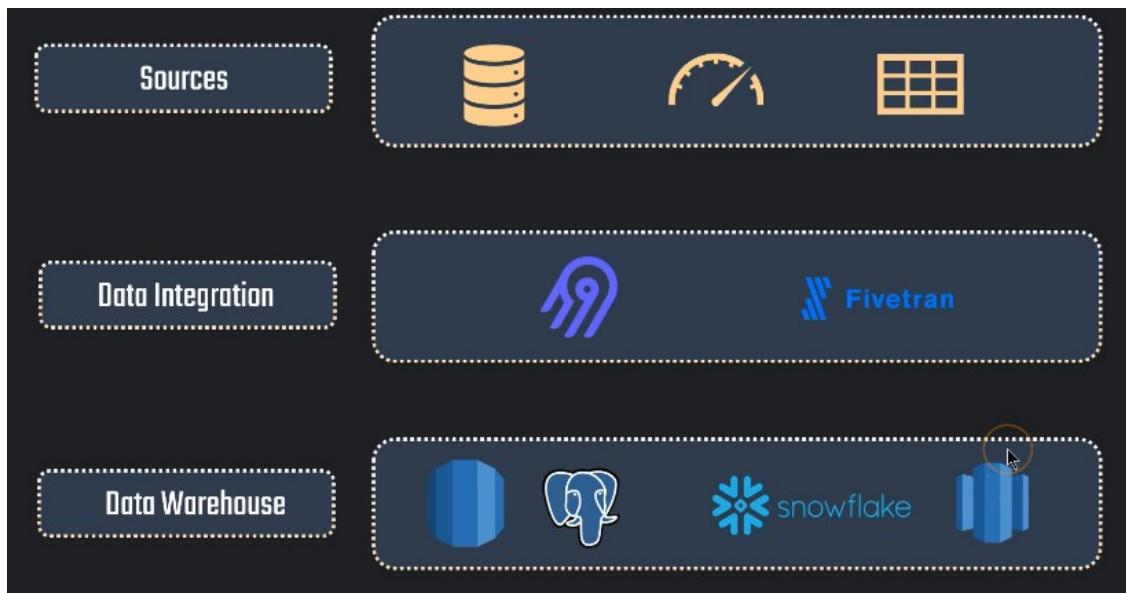
Procesamiento: Apache Spark ← (aquí está)

Orquestación: Airflow

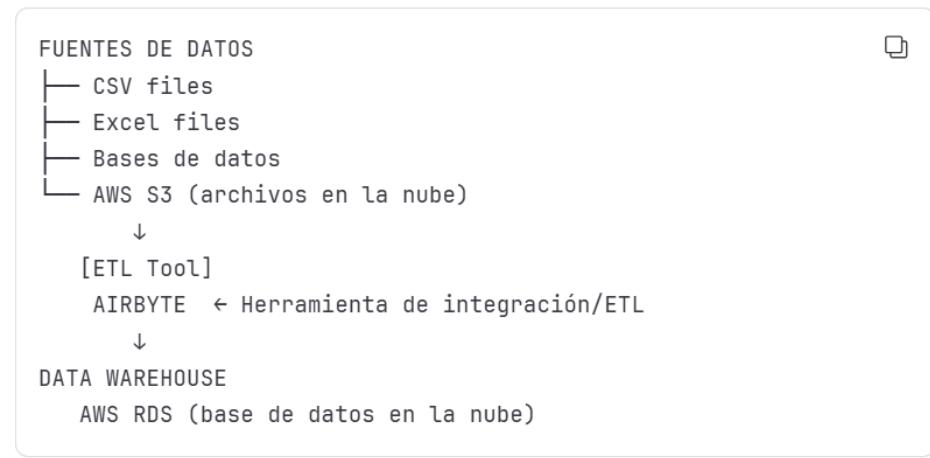
Resumen:

- Big Data SIN Spark → Posible pero lento
- **Spark = Creado PARA Big Data**

1.5 Comprendión de la configuración de nuestra herramienta de almacenamiento de datos



Stack que usaremos (EJEMPLOS)



Herramientas Específicas

1. Fuentes de Datos

- Archivos CSV/Excel/texto
- Bases de datos
- S3 buckets (AWS)

2. ETL: Airbyte

¿Por qué Airbyte?

- Fácil de usar
- Fácil de configurar
- Versión gratuita
- Moderno

Alternativas: Existen muchas otras (Fivetran, Talend, etc.)

3. Data Warehouse: AWS RDS

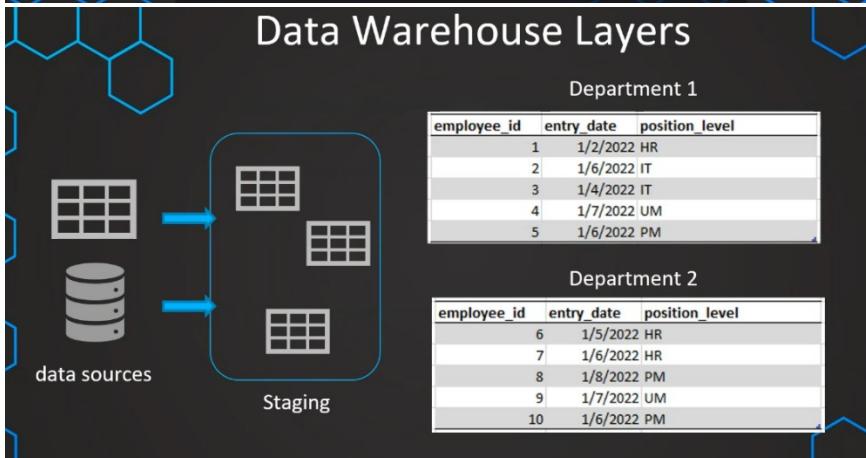
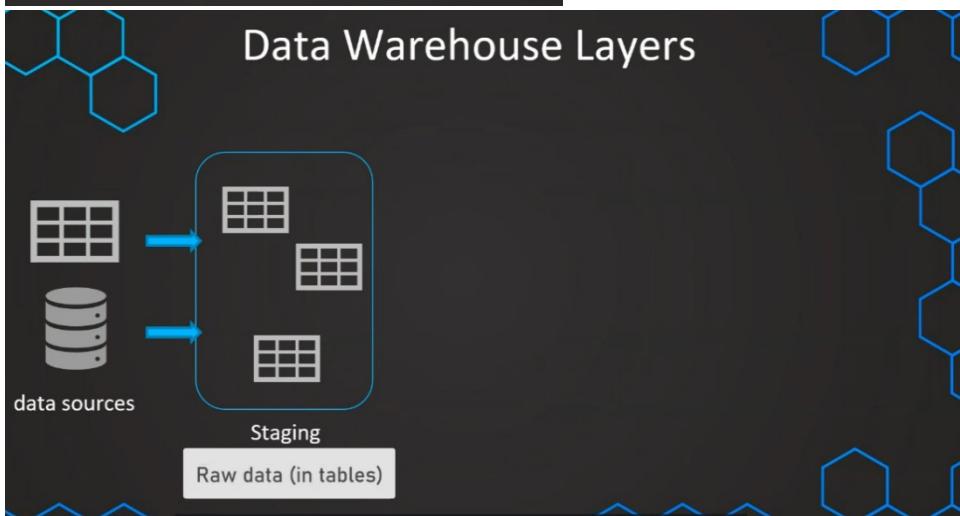
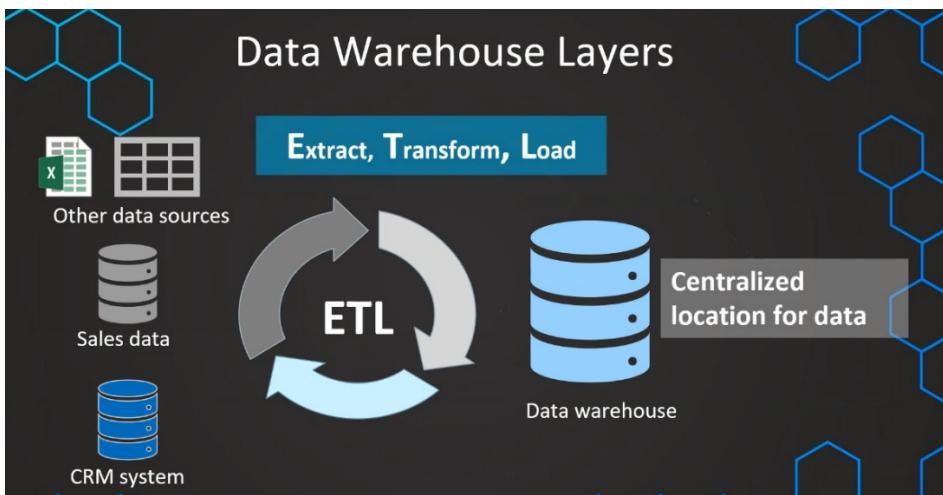
Base de datos en la nube (AWS)

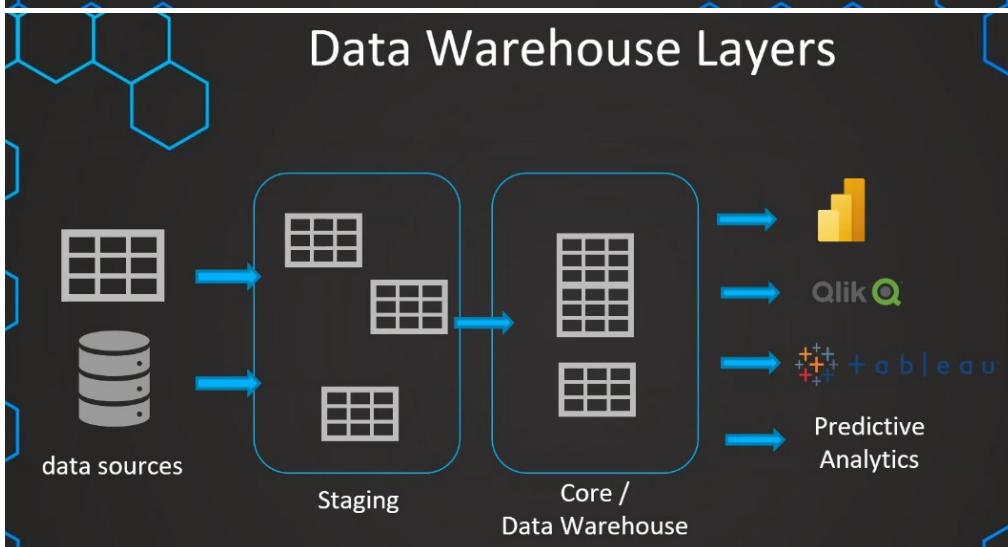
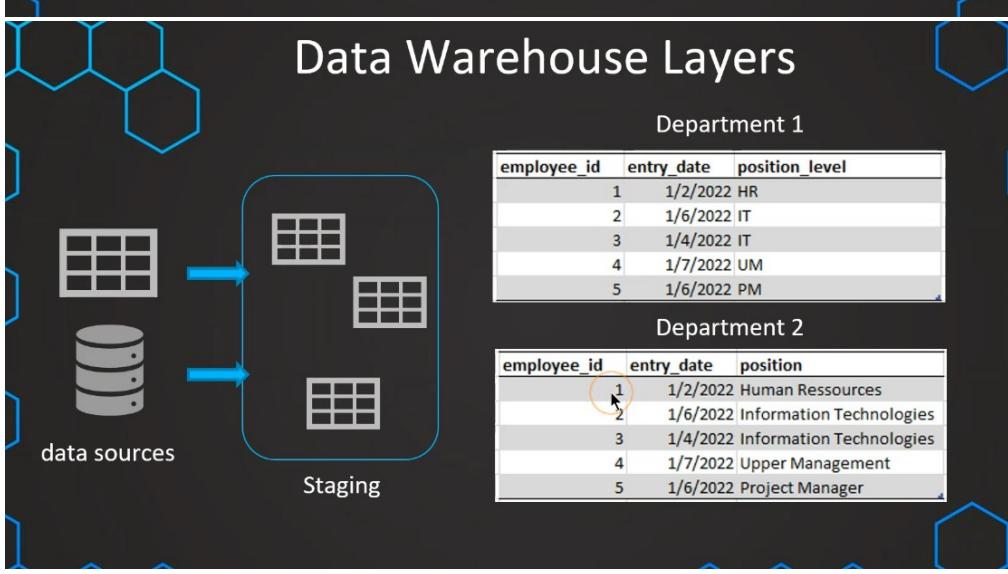
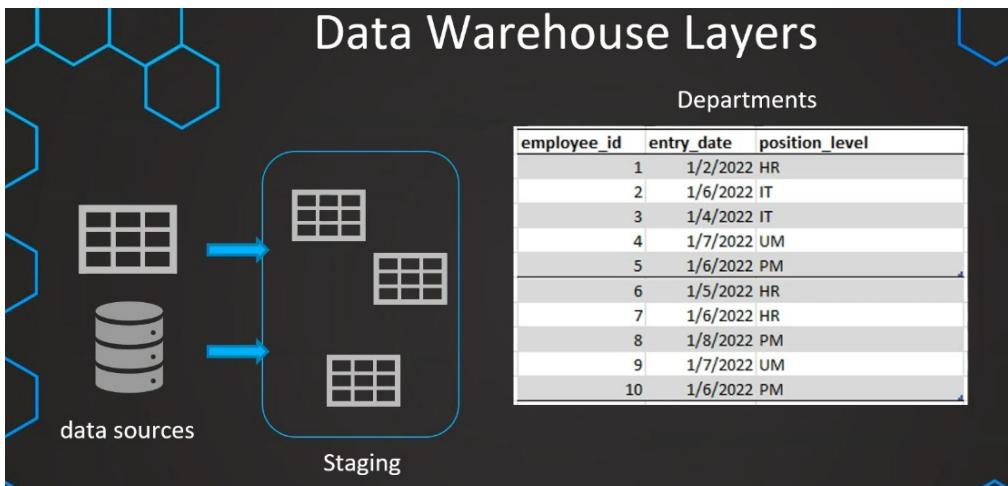
- PostgreSQL
- MySQL
- Otros motores

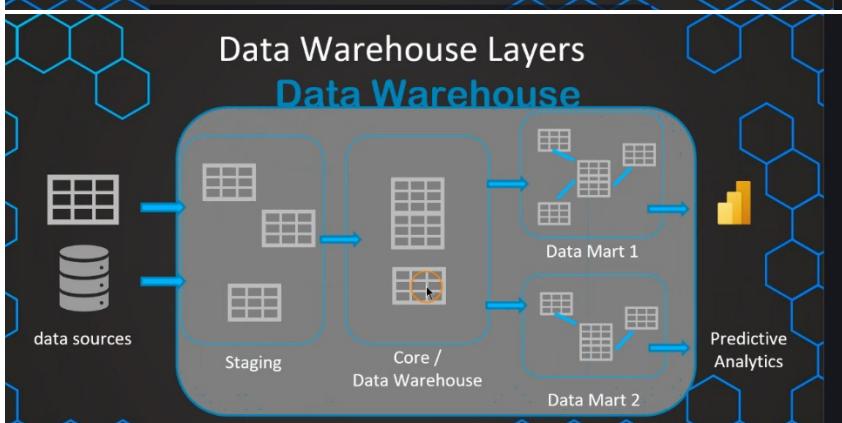
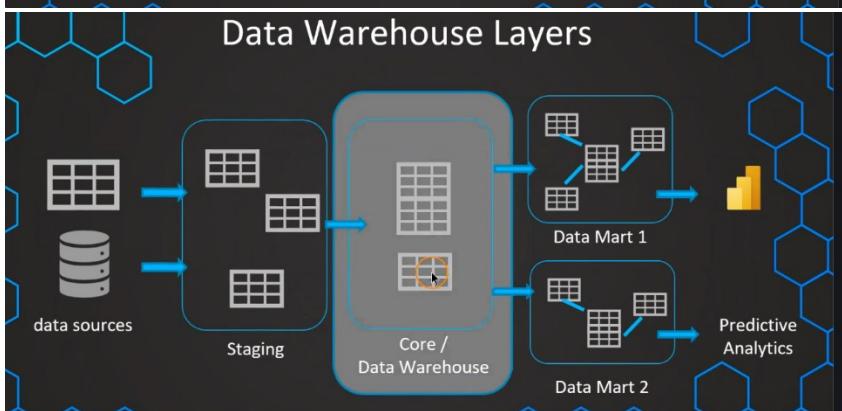
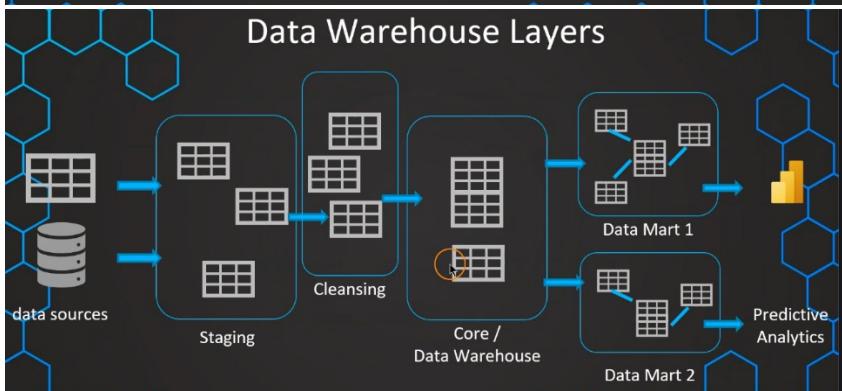
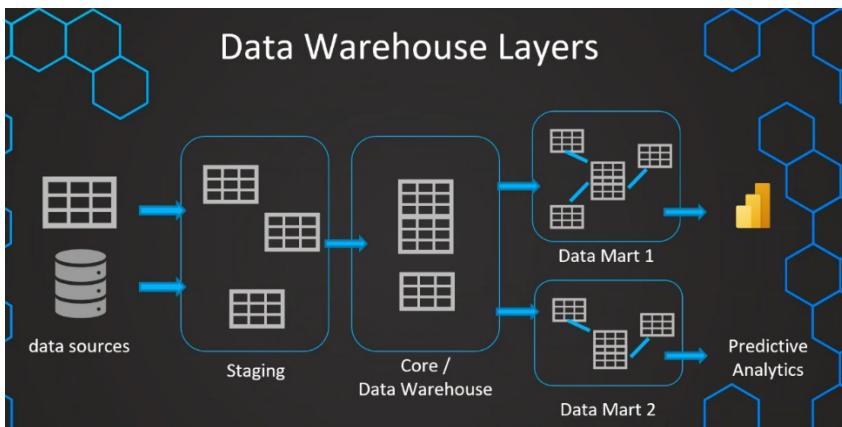
Alternativas especializadas (mayor rendimiento):

- Snowflake
- AWS Redshift
- Google BigQuery

2. Data Warehouse Architecture (Theory)







2.1 Capas de un Data Warehouse

Arquitectura Completa

Un Data Warehouse NO es una sola base de datos, sino un **sistema de capas** donde cada una tiene un propósito específico.

2.1.1 Las 4 Capas Principales

2.1.1.1: STAGING (Área de Preparación)

Propósito: Recibir datos RAW sin transformar.

Características:

- Datos tal cual vienen de las fuentes
- Mínima o nula transformación
- Formato tabular (todo convertido a tablas)
- Datos temporales (se sobreescreiben)

Ejemplo Práctico:

Fuentes originales:

- employees_dept1.csv
- employees_dept2.xlsx
- employees_dept3.sql

↓ ETL extrae a Staging ↓

Staging contiene:

- staging.employees_dept1 (tabla)
- staging.employees_dept2 (tabla)
- staging.employees_dept3 (tabla)

Transformaciones mínimas permitidas:

- Convertir Excel → Tabla SQL
- Append/Union de tablas similares
- NO se limpian datos
- NO se estandarizan nombres

2.1.1.2: CLEANSING (Limpieza) - OPCIONAL

Propósito: Limpiar datos muy sucios.

Cuándo usarla:

- Datos con muchos errores
- Formatos inconsistentes
- Necesitas validación intensiva

Ejemplo Práctico:

```
sql
-- Staging tiene esto:
employee_id | entry_date      | position_level
1           | "01/02/2022"    | "HR"
2           | "2022-06-01"    | "IT"          -- espacio extra
3           | NULL            | "it"          -- minúscula

-- Cleansing corrige:
employee_id | entry_date      | position_level
1           | 2022-02-01     | "HR"
2           | 2022-06-01     | "IT"
3           | 2022-01-01     | "IT"          -- fecha default
```

Nota: Muchos DWH modernos NO usan esta capa, hacen limpieza directo en la transformación Staging → Core.

2.1.1.3: CORE / DATA WAREHOUSE - CORAZÓN DEL SISTEMA ❤️

Propósito: Datos limpios, integrados, estandarizados.

Características:

- Única fuente de verdad (Single Source of Truth)
- Datos históricos completos
- Schema consistente
- Optimizado para consultas

Ejemplo Práctico:

```
Problema en Staging:  
Department 1:  
employee_id | position_level  
1           | "HR"  
2           | "IT"  
  
Department 2:  
employee_id | position      -- columna diferente  
1           | "Human Resources" -- nombre diferente  
2           | "Information Technologies"  
  
↓ Transformación a Core ↓  
  
Core tiene:  
employee_id | department_id | position_standardized | entry_date  
1           | 1                 | "Human Resources"    | 2022-02-01  
2           | 1                 | "Information Tech"  | 2022-06-01  
1           | 2                 | "Human Resources"    | 2022-05-01  
2           | 2                 | "Information Tech"  | 2022-06-01
```

Transformaciones en Core:

- Estandarización de nombres
- Unificación de IDs
- Joins entre tablas
- Agregaciones
- Modelado dimensional (Star Schema, Snowflake)

2.1.1.4: DATA MARTS - OPCIONAL

Propósito: Subconjuntos especializados para equipos específicos.

Cuándo usarlos:

- DWH muy grande (100+ tablas)
- Equipos con necesidades muy diferentes
- Optimizar rendimiento

Ejemplo Práctico:

Core tiene 150 tablas:

- Ventas (50 tablas)
- RR.HH. (30 tablas)
- Finanzas (40 tablas)
- Marketing (30 tablas)

↓ Crear Data Marts ↓

Data Mart Ventas (solo 50 tablas relevantes):

- dim_customers
 - dim_products
 - fact_sales
 - dim_dates
- Equipo ventas consulta SOLO estas tablas

Data Mart RR.HH. (solo 30 tablas):

- dim_employees
 - dim_departments
 - fact_payroll
- Equipo RR.HH. consulta SOLO estas

Ventajas Data Marts:

- Más rápido (menos tablas)
- Más fácil de usar
- Puede usar tecnología especializada (in-memory, cubos OLAP)

2.1.2 Flujo Completo con Ejemplo Real

Caso: E-commerce con 3 fuentes de datos

PASO 1: STAGING

Fuente: orders_system.db (PostgreSQL)

Fuente: products.csv (archivo CSV)

Fuente: customers_api.json (API REST)

↓ ETL extrae RAW ↓

Staging:

- staging.orders_raw (sin transformar)
- staging.products_raw (sin transformar)
- staging.customers_raw (sin transformar)

PASO 2: CLEANSING (opcional)

Limpia:

- Fechas inconsistentes
- Emails duplicados
- Precios negativos

PASO 3: CORE / DWH

Transforma y modela:

dim_customers (dimensión)

- customer_id (PK)
- customer_name
- email
- country
- registration_date

dim_products (dimensión)

- product_id (PK)
- product_name
- category
- price

dim_dates (dimensión)

- date_id (PK)
- date
- year
- month
- quarter

fact_sales (hechos)

- sale_id (PK)
- customer_id (FK)
- product_id (FK)
- date_id (FK)
- quantity
- total_amount

→ Star Schema optimizado para consultas

PASO 4: DATA MARTS

Data Mart "Marketing":

- Copia solo: dim_customers, dim_dates, fact_sales
- Agrega métricas pre-calculadas
- Optimiza para dashboards

Data Mart "Inventario":

- Copia solo: dim_products, dim_dates, fact_inventory
- Optimiza para reportes de stock

2.1.3 Resumen: ¿Qué es el "Data Warehouse"? – 2 Perspectivas – Tabla Comparativa

Dos perspectivas:

Perspectiva Técnica:

El Data Warehouse = **TODAS LAS CAPAS JUNTAS**

- Staging
- Cleansing (opcional)
- Core
- Data Marts (opcional)

Perspectiva Usuario Final:

El Data Warehouse = **Core + Data Marts**

Porque son las capas que ellos ven y usan directamente.

Tabla Comparativa de Capas

Capa	Propósito	Datos	Usuarios	Transformación
Staging	Recibir datos	RAW	Solo ETL	Mínima
Cleansing	Limpiar	Semi-limpio	Solo ETL	Limpieza
Core	Integrar	Limpio	Analistas	Completa
Data Marts	Especializar	Optimizado	Equipos	Agregación

2.1.4 Ejemplo Código SQL + Analogía

Staging → Core (transformación)

```
● ● ●

-- STAGING (raw data)
SELECT * FROM staging.employees_dept1;
-- Resultado:
-- employee_id | entry_date      | position_level
-- 1           | "1/2/2022"     | "HR"
-- 2           | "1/6/2022"     | "IT"

SELECT * FROM staging.employees_dept2;
-- Resultado:
-- employee_id | entry_date      | position
-- 1           | "1/5/2022"     | "Human Resources"
-- 2           | "1/6/2022"     | "Information Technologies"

-- TRANSFORMACIÓN A CORE
INSERT INTO core.dim_employees (
    employee_id_unified,
    department_id,
    position_standardized,
    entry_date
)
SELECT
    CONCAT(department_id, '-', employee_id) AS employee_id_unified,
    1 AS department_id,
    CASE
        WHEN position_level = 'HR' THEN 'Human Resources'
        WHEN position_level = 'IT' THEN 'Information Technology'
        ELSE position_level
    END AS position_standardized,
    TO_DATE(entry_date, 'MM/DD/YYYY') AS entry_date
FROM staging.employees_dept1
UNION ALL
SELECT
    CONCAT(department_id, '-', employee_id) AS employee_id_unified,
    2 AS department_id,
    CASE
        WHEN position = 'Human Resources' THEN 'Human Resources'
        WHEN position = 'Information Technologies' THEN 'Information Technology'
        ELSE position
    END AS position_standardized,
    TO_DATE(entry_date, 'MM/DD/YYYY') AS entry_date
FROM staging.employees_dept2;

-- RESULTADO EN CORE
SELECT * FROM core.dim_employees;
-- employee_id_unified | department_id | position_standardized | entry_date
-- 1-1                 | 1           | Human Resources     | 2022-01-02
-- 1-2                 | 1           | Information Technology | 2022-01-06
-- 2-1                 | 2           | Human Resources     | 2022-01-05
-- 2-2                 | 2           | Information Technology | 2022-01-06
```

Best Practices

1. **Staging:** NUNCA borrar datos históricos sin backup
2. **Core:** Implementar slowly changing dimensions (SCD)
3. **Data Marts:** Actualizar con frecuencia (diario/semanal)
4. **Cleansing:** Documentar TODAS las reglas de limpieza

Analogía

Data Warehouse = Fábrica de productos

- **Staging** = Muelle de carga (materia prima sin procesar)
- **Cleansing** = Control de calidad (inspección)
- **Core** = Línea de producción (transformación)
- **Data Marts** = Bodegas especializadas (productos finales organizados)

2.1.4.1 Explicación Paso a Paso del Código SQL

PARTE 1: STAGING (Datos RAW)

Consulta 1:



```
SELECT * FROM staging.employees_dept1;
```

¿Qué hace?

Muestra los datos RAW (sin procesar) del Departamento 1.

Resultado:

...

employee_id	entry_date	position_level
1	"1/2/2022"	"HR"
2	"1/6/2022"	"IT"

Problemas detectados:

- Fecha en formato texto (no DATE)
- Abreviaciones ("HR", "IT")
- IDs repetidos entre departamentos (ID=1 existe en Dept1 y Dept2)

Consulta 2:



```
SELECT * FROM staging.employees_dept2;
```

...

¿Qué hace?

Muestra los datos RAW del Departamento 2.

Resultado:

...

employee_id	entry_date	position
1	"1/5/2022"	"Human Resources"
2	"1/6/2022"	"Information Technologies"

Problemas detectados:

- Columna diferente: position vs position_level
- Nombres completos vs abreviaciones
- IDs repetidos (ID=1 y 2 se repiten)

PARTE 2: TRANSFORMACIÓN (Staging → Core)



```
INSERT INTO core.dim_employees (
    employee_id_unified,
    department_id,
    position_standardized,
    entry_date
)
```

¿Qué hace? Declara que vamos a INSERTAR datos en la tabla core.dim_employees con estas columnas:

- employee_id_unified → ID Único global
- department_id → Identificador del departamento
- position_standardized → Cargo estandarizado
- entry_date → Fecha de ingreso (formato DATE)

Primera parte del SELECT (Dept 1):



```
SELECT
    CONCAT(department_id, '-', employee_id) AS employee_id_unified,
```

¿Qué hace?

Crea un ID ÚNICO combinando departamento + employee_id.

Ejemplo:

```

Dept 1, employee\_id = 1 → "1-1"

Dept 1, employee\_id = 2 → "1-2"

---

**¿Por qué?** Porque el **employee\_id** se repite entre departamentos (ambos tienen 1 y 2).



```
1 AS department_id,
```

```

¿Qué hace?

Asigna el valor fijo `1` (Departamento 1) a TODOS los registros de esta tabla.

Resultado:

```

Todos los empleados de dept1 tendrán department\_id = 1



CASE

```
WHEN position_level = 'HR' THEN 'Human Resources'
WHEN position_level = 'IT' THEN 'Information Technology'
ELSE position_level
END AS position_standardized,
...
```

\*\*¿Qué hace?\*\*

ESTANDARIZA los nombres de cargos usando `CASE`:

\*\*Lógica:\*\*

...

```
SI position_level = 'HR'
ENTONCES escribe 'Human Resources'
SI position_level = 'IT'
ENTONCES escribe 'Information Technology'
SI NO (cualquier otro valor)
ENTONCES deja el valor original
...
```

\*\*Transformación:\*\*

...

"HR" → "Human Resources"

"IT" → "Information Technology"



```
TO_DATE(entry_date, 'MM/DD/YYYY') AS entry_date
```

...

\*\*¿Qué hace?\*\*

Convierte el texto `1/2/2022` a un tipo de dato \*\*DATE\*\*.

\*\*Transformación:\*\*

...

"1/2/2022" (texto) → 2022-01-02 (fecha)

"1/6/2022" (texto) → 2022-01-06 (fecha)

¿Por qué? Las fechas en texto NO se pueden usar para:

- Ordenar cronológicamente
- Calcular diferencias
- Filtrar por rangos



```
FROM staging.employees_dept1
```

¿Qué hace?: Especifica que los datos vienen de la tabla **staging.employees\_dept1**.

## UNION ALL:



UNION ALL

¿Qué hace? COMBINA los resultados del SELECT anterior CON el siguiente SELECT.

Diferencia UNION vs UNION ALL:

- UNION → Elimina duplicados
- UNION ALL → Mantiene TODO (más rápido)

En este caso usamos UNION ALL porque sabemos que NO hay duplicados (son departamentos diferentes).

## Segunda parte del SELECT (Dept 2):



SELECT

```
CONCAT(department_id, '-', employee_id) AS employee_id_unified,
2 AS department_id, -- Departamento 2
CASE
 WHEN position = 'Human Resources' THEN 'Human Resources'
 WHEN position = 'Information Technologies' THEN 'Information Technology'
 ELSE position
END AS position_standardized,
TO_DATE(entry_date, 'MM/DD/YYYY') AS entry_date
FROM staging.employees_dept2;
```

¿Qué hace? EXACTAMENTE lo mismo que el SELECT anterior, PERO:

- Lee de staging.employees\_dept2
- Asigna department\_id = 2
- Usa la columna position (no position\_level)
- Estandariza "Information Technologies" → "Information Technology"

## PARTE 3: RESULTADO FINAL

```
SELECT * FROM core.dim_employees;
```

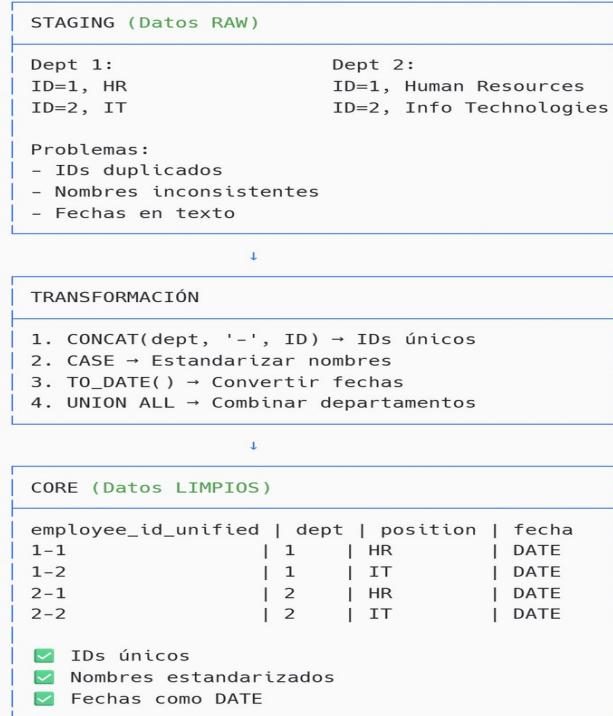
\*\*¿Qué hace?\*\*  
Muestra el resultado FINAL después de la transformación.

\*\*Resultado:\*\*

```
employee_id_unified | department_id | position_standardized | entry_date
1-1 | 1 | Human Resources | 2022-01-02
1-2 | 1 | Information Technology | 2022-01-06
2-1 | 2 | Human Resources | 2022-01-05
2-2 | 2 | Information Technology | 2022-01-06
```

---

```
🔍 Flujo Completo Visualizado
```



## 📈 Resumen Breve

\*\*El código hace 3 cosas:\*\*

1. \*\*Lee\*\* datos RAW de 2 tablas staging (**Dept 1 y Dept 2**)
2. \*\*Transforma\*\*:
  - Crea IDs únicos (1-1, 1-2, 2-1, 2-2)
  - Estandariza nombres de cargos
  - Convierte fechas texto → DATE
3. \*\*Inserta\*\* en la tabla Core limpia y unificada

\*\*Antes vs Despues:\*\*

**ANTES (Staging):**

- 2 tablas separadas
- Datos inconsistentes
- IDs duplicados

**DESPUÉS (Core):**

- 1 tabla unificada
- Datos estandarizados
- IDs únicos

## 2.1.4.2 Explicacion de los campos redundantes de: employee\_id\_unified y department\_id

pero el 1-1 (que es el primer 1 y el segundo 1) al final si el primer 1 es 1.  
entonces el department id si o si es 1?

Sí, tienes razón:



```
employee_id_unified = "1-1"
 ↑ ↑
 | |
 | └ employee_id original
 | |
 | └ department_id
```

Si el primer "1" es el department\_id,  
entonces department\_id SIEMPRE será 1

Análisis:



|                     |  |                                                |
|---------------------|--|------------------------------------------------|
| employee_id_unified |  | department_id                                  |
| 1-1                 |  | 1                    ← El primer "1" se repite |
| 1-2                 |  | 1                    ← El primer "1" se repite |
| 2-1                 |  | 2                    ← El primer "2" se repite |
| 2-2                 |  | 2                    ← El primer "2" se repite |

La columna **department\_id** es **REDUNDANTE** porque ya está en el **employee\_id\_unified**.

¿Por qué se hace así entonces?

Razones prácticas:

1. Para hacer JOINs más fáciles:



```
-- CON department_id (fácil)
SELECT *
FROM dim_employees e
JOIN dim_departments d ON e.department_id = d.department_id

-- SIN department_id (complicado)
SELECT *
FROM dim_employees e
JOIN dim_departments d ON SPLIT_PART(e.employee_id_unified, '-', 1) = d.department_id
```

2. Para filtros más rápidos:



```
-- CON department_id (rápido - usa índice)
WHERE department_id = 1

-- SIN department_id (lento - necesita parsear string)
WHERE employee_id_unified LIKE '1-%'
```

### **3. Normalización de base de datos: Separar el ID compuesto en sus partes permite crear relaciones (Foreign Keys).**

---

#### **Resumen:**

Sí, es redundante, pero se mantiene por:

- Performance (índices, joins)
- Simplicidad (no parsear strings)
- Normalización (Foreign Keys)

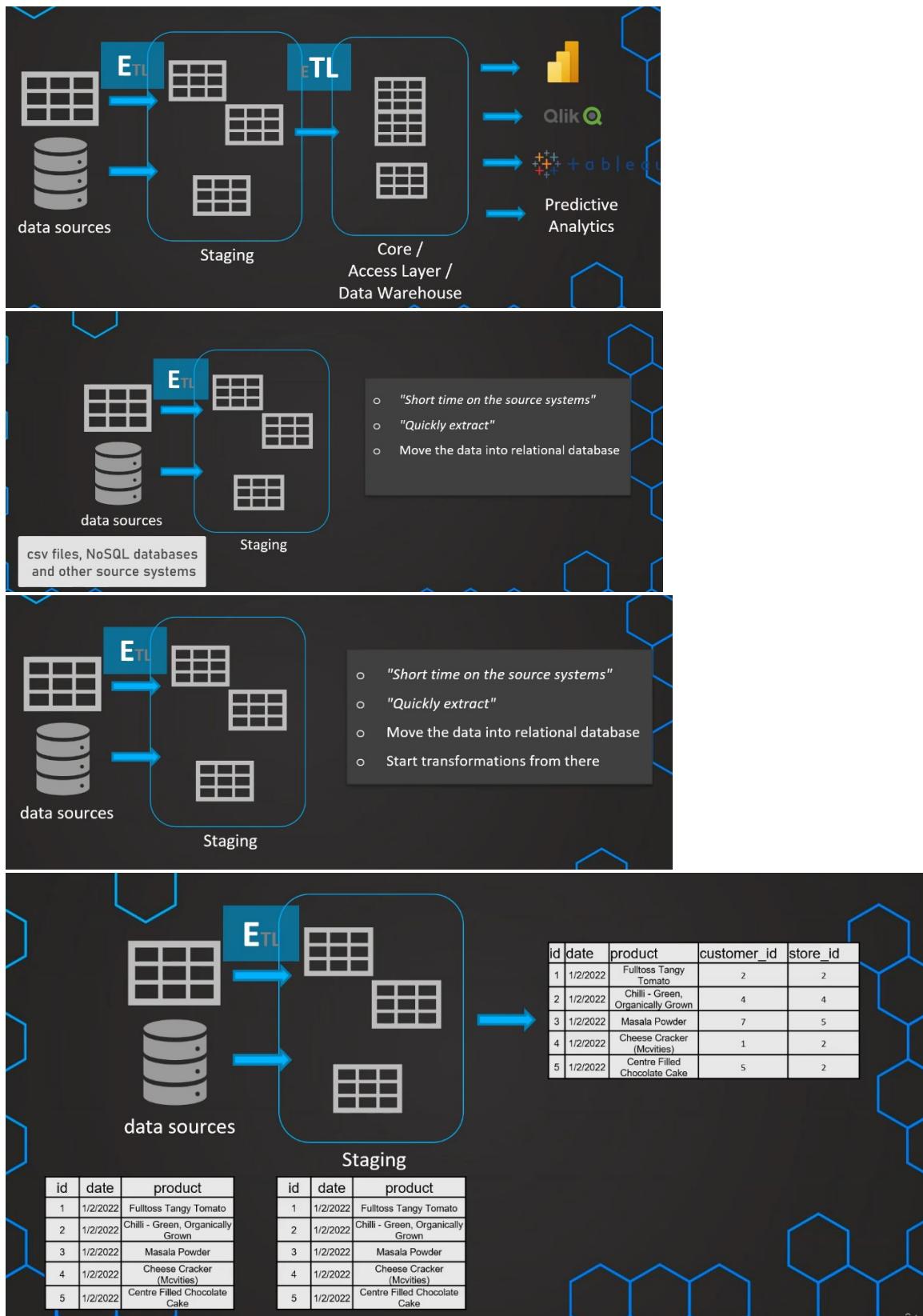
Es una decisión de **diseño de base de datos** común en Data Warehouses.

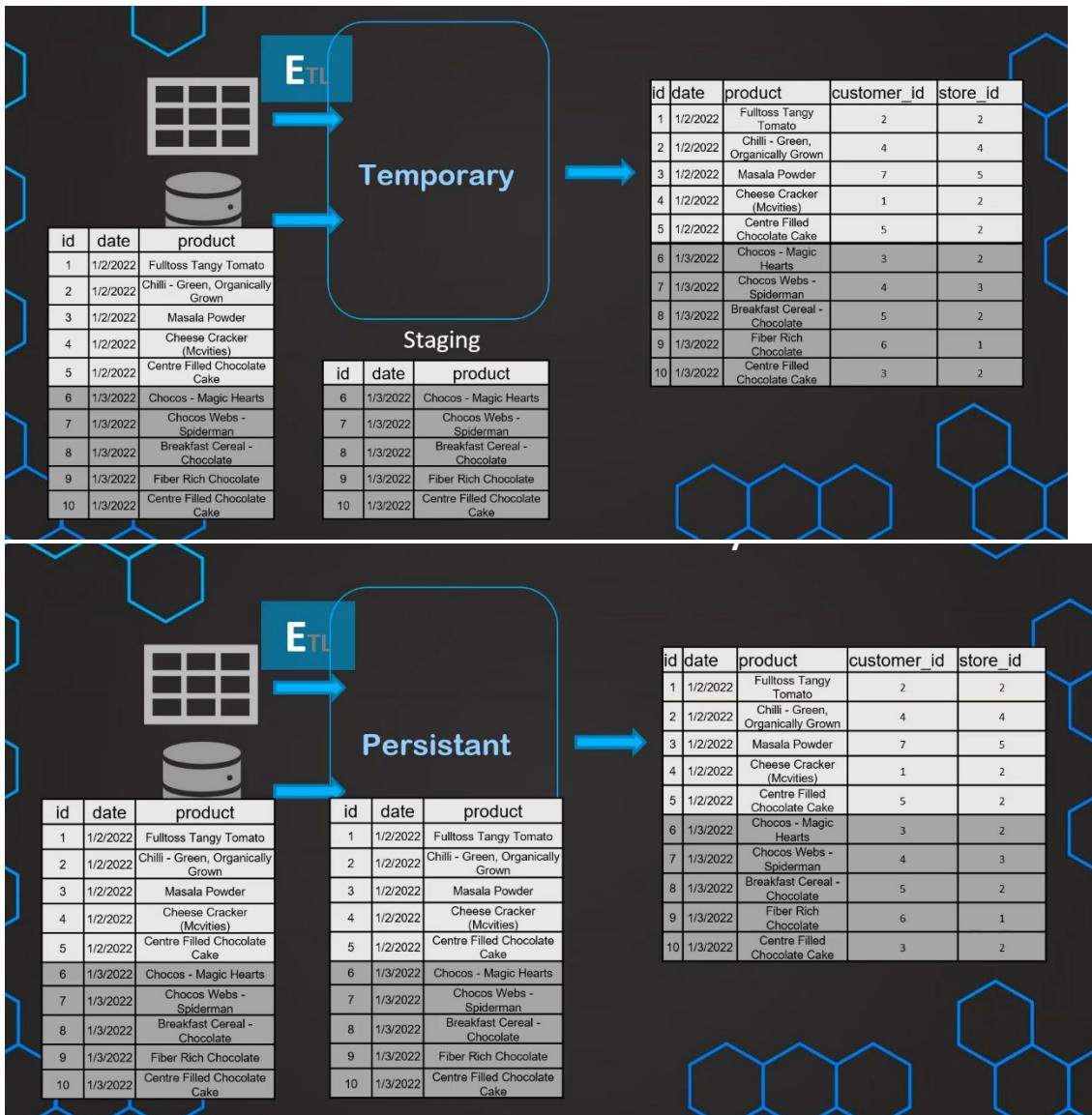
#### **2.1.5 Conclusión**

Un Data Warehouse NO es una sola tabla gigante. Es un **sistema arquitectónico de capas**, donde cada capa tiene un rol específico en el flujo de datos desde las fuentes hasta los usuarios finales.

La magia está en la **transformación progresiva** de datos raw → datos limpios → datos optimizados para análisis.

## 2.2: Staging Area





- ✓ **Staging Layer is the landing zone extracted data**
- ✓ **Data in tables and on a separate database**
- ✓ **As little "touching" as possible**
- ✓ **We don't change the source systems**
- ✓ **Temporary or Persistent Staging Layers**

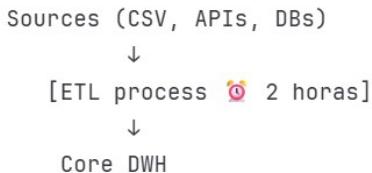
## 2.2.1 ¿Qué es la Staging Layer?

La **Staging Layer** es la **zona de aterrizaje** donde llegan los datos RAW extraídos de las fuentes ANTES de ser transformados.

**Analogía:** Es como el muelle de carga de una fábrica donde llegan las materias primas sin procesar.

## 2.2.2 ¿Por qué necesitamos Staging? + Solucion con Staging

**Problema sin Staging:**

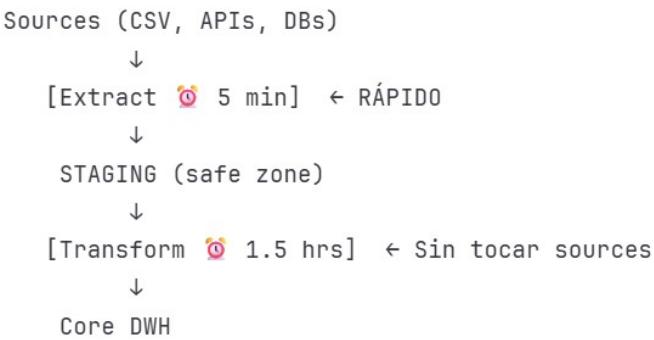


**Riesgos:**

- ✗ 2 horas conectados a sistemas productivos
- ✗ Consumir recursos de sistemas críticos
- ✗ Si falla la transformación, hay que volver a extraer TODO

---

## Solución con Staging:



**Beneficios:**

- ✅ Extracción rápida (5-10 min vs horas)
- ✅ No sobrecargamos sistemas productivos
- ✅ Datos seguros para transformar
- ✅ Si falla, reintentamos desde Staging

## **2.2.3 Propósito de Staging – 3 razones principales**

**3 razones principales:**

### **1. Tiempo mínimo en sistemas fuente**

Systems fuente = Sistemas productivos críticos  
(e.g., sistema de ventas, CRM, ERP)

NO queremos ralentizarlos con consultas pesadas

### **2. Convertir TODO a tablas relacionales**

CSV files → Staging tables  
JSON APIs → Staging tables  
NoSQL DBs → Staging tables  
Excel files → Staging tables



Ahora TODO está en SQL → Fácil de transformar

### **3. Aplicar transformaciones desde zona segura**

Staging (datos seguros) → Transform → Core

## 2.2.4 Ejemplo Práctico: Sistema de Ventas

### Día 1: Primera carga

#### PASO 1: Extract (fuente → staging)

```
Sistema fuente (producción):
id | date | product
1 | 1/2/2022 | Fulltoss Tangy Tomato
2 | 1/2/2022 | Chilli - Green
3 | 1/2/2022 | Masala Powder
4 | 1/2/2022 | Cheese Cracker
5 | 1/2/2022 | Centre Filled Chocolate Cake
```

↓ ETL extrae RÁPIDO (30 segundos) ↓

```
Staging:
id | date | product
1 | 1/2/2022 | Fulltoss Tangy Tomato
2 | 1/2/2022 | Chilli - Green
3 | 1/2/2022 | Masala Powder
4 | 1/2/2022 | Cheese Cracker
5 | 1/2/2022 | Centre Filled Chocolate Cake
```

#### PASO 2: Transform (staging → core)

```
sql
-- Transformaciones aplicadas:
-- - Join con dim_customers
-- - Join con dim_stores
-- - Estandarizar nombres

Core DWH:
id | date | product | customer_id | store_id
1 | 2022-01-02 | Fulltoss Tangy Tomato | 2 | 2
2 | 2022-01-02 | Chilli Green Organic | 4 | 4
3 | 2022-01-02 | Masala Powder | 7 | 5
4 | 2022-01-02 | Cheese Cracker | 1 | 2
5 | 2022-01-02 | Centre Filled Choco Cake | 5 | 2
```

### Día 2: Carga incremental

Ahora tenemos **2 opciones** según el tipo de Staging:

## 2.2.4.1 OPCIÓN 1: TEMPORARY STAGING (Temporal) (2 opciones)

### Características:

- Se **TRUNCA** (borra) después de cada carga
- Solo contiene datos del ciclo actual
- Más común (80% de casos)

---

### Flujo Día 2 con Temporary Staging:

#### PASO 1: Truncar Staging

```
sql
TRUNCATE TABLE staging.sales; -- Staging queda VACÍA
```

#### PASO 2: Identificar datos nuevos (Delta Logic)

Necesitamos saber QUÉ es nuevo en la fuente.

#### Opción A: Usar columna ID

```
sql
-- Último ID cargado fue 5
SELECT * FROM source.sales WHERE id > 5;

Resultado:
id | date | product
6 | 1/3/2022 | Chocos - Magic Hearts
7 | 1/3/2022 | Chocos Webs - Spiderman
8 | 1/3/2022 | Breakfast Cereal - Chocolate
9 | 1/3/2022 | Fiber Rich Chocolate
10 | 1/3/2022 | Centre Filled Chocolate Cake
```

**⚠ Problema con ID:** Si el ID NO es estrictamente incremental (puede saltar números, duplicarse), cargamos datos incorrectos.

---

#### Opción B: Usar columna DATE (MÁS COMÚN)

```
sql
-- Última fecha cargada fue 1/2/2022
SELECT * FROM source.sales WHERE date > '2022-01-02';

Resultado:
id | date | product
6 | 1/3/2022 | Chocos - Magic Hearts
7 | 1/3/2022 | Chocos Webs - Spiderman
8 | 1/3/2022 | Breakfast Cereal - Chocolate
9 | 1/3/2022 | Fiber Rich Chocolate
10 | 1/3/2022 | Centre Filled Chocolate Cake
```

**✓ Mejor:** La fecha es más confiable.

## PASO 3: Extract nuevos datos a Staging

```
Staging (limpio después de TRUNCATE):
id | date | product
6 | 1/3/2022 | Chocos - Magic Hearts
7 | 1/3/2022 | Chocos Webs - Spiderman
8 | 1/3/2022 | Breakfast Cereal - Chocolate
9 | 1/3/2022 | Fiber Rich Chocolate
10 | 1/3/2022 | Centre Filled Chocolate Cake
```

## PASO 4: Transform y APPEND a Core

```
sql

INSERT INTO core.fact_sales (...)
SELECT ... FROM staging.sales
JOIN dim_customers ...
JOIN dim_stores ...;

Core DWH (con nuevos datos):
id | date | product | customer_id | store_id
1 | 2022-01-02 | Fulltoss Tangy Tomato | 2 | 2
2 | 2022-01-02 | Chilli Green Organic | 4 | 4
...
5 | 2022-01-02 | Centre Filled Choco Cake | 5 | 2
6 | 2022-01-03 | Chocos Magic Hearts | 3 | 2 ← NUEVO
7 | 2022-01-03 | Chocos Webs Spiderman | 4 | 3 ← NUEVO
8 | 2022-01-03 | Breakfast Cereal Choco | 5 | 2 ← NUEVO
9 | 2022-01-03 | Fiber Rich Chocolate | 6 | 1 ← NUEVO
10 | 2022-01-03 | Centre Filled Choco Cake | 3 | 2 ← NUEVO
```

## PASO 5: Truncar Staging de nuevo

```
sql

TRUNCATE TABLE staging.sales; -- Listo para mañana
```

## Ventajas Temporary Staging:

- ✓ Ahorra espacio (solo datos actuales)
- ✓ Staging siempre limpio
- ✓ Más rápido (menos datos)

## Desventajas:

- ✗ Si transformación falla, hay que volver a extraer de source
- ✗ No puedes "retroceder en el tiempo" fácilmente

## 2.2.4.2 OPCION 2: PERSISTENT STAGING (Persistente) – capa rara

Características:

- **NUNCA** se trunca
- Acumula **TODOS** los datos históricos
- Menos común (20% de casos)

---

Flujo Día 2 con Persistent Staging:

### PASO 1: NO truncar (mantener histórico)

```
Staging (mantiene datos previos):
```

| id | date     | product                      |
|----|----------|------------------------------|
| 1  | 1/2/2022 | Fulltoss Tangy Tomato        |
| 2  | 1/2/2022 | Chilli - Green               |
| 3  | 1/2/2022 | Masala Powder                |
| 4  | 1/2/2022 | Cheese Cracker               |
| 5  | 1/2/2022 | Centre Filled Chocolate Cake |

### PASO 2: Extract nuevos datos (Delta Logic igual)

```
sql
```

```
SELECT * FROM source.sales WHERE date > '2022-01-02';
```

### PASO 3: APPEND a Staging (no reemplazar)

```
sql
```

```
INSERT INTO staging.sales
SELECT * FROM source.sales WHERE date > '2022-01-02';
```

Staging (acumulativo):

| id | date     | product                 | ← DÍA | ← NUEVO |
|----|----------|-------------------------|-------|---------|
| 1  | 1/2/2022 | Fulltoss Tangy Tomato   | 1     |         |
| 2  | 1/2/2022 | Chilli - Green          | 1     |         |
| 3  | 1/2/2022 | Masala Powder           | 1     |         |
| 4  | 1/2/2022 | Cheese Cracker          | 1     |         |
| 5  | 1/2/2022 | Centre Filled Chocolate | 1     |         |
| 6  | 1/3/2022 | Chocos - Magic Hearts   | 2     | NUEVO   |
| 7  | 1/3/2022 | Chocos Webs - Spiderman | 2     | NUEVO   |
| 8  | 1/3/2022 | Breakfast Cereal        | 2     | NUEVO   |
| 9  | 1/3/2022 | Fiber Rich Chocolate    | 2     | NUEVO   |
| 10 | 1/3/2022 | Centre Filled Chocolate | 2     | NUEVO   |

### PASO 4: Transform solo nuevos datos

sql



```
INSERT INTO core.fact_sales (...)
SELECT ... FROM staging.sales
WHERE date > '2022-01-02' -- Solo procesar nuevos
JOIN dim_customers ...;
```

### Ventajas Persistent Staging:

- Backup de datos raw siempre disponible
- Fácil "retroceder en el tiempo" si hay errores
- No necesitas volver al source system

### Desventajas:

- Consumo MÁS espacio (todo el histórico)
- Más lento (más datos que gestionar)
- Requiere buena gestión de datos duplicados

### 2.2.4.3 Comparación: Temporary vs Persistent + Cuando usar cada tipo

| Característica       | Temporary                                                      | Persistent                                                     |
|----------------------|----------------------------------------------------------------|----------------------------------------------------------------|
| Se trunca            | <input checked="" type="checkbox"/> Sí (cada ciclo)            | <input checked="" type="checkbox"/> No (nunca)                 |
| Datos históricos     | <input checked="" type="checkbox"/> No                         | <input checked="" type="checkbox"/> Sí (todos)                 |
| Espacio usado        | <input checked="" type="checkbox"/> Bajo                       | <input checked="" type="checkbox"/> Alto                       |
| Velocidad            | <input checked="" type="checkbox"/> Rápido                     | <input checked="" type="checkbox"/> Medio                      |
| Recuperación errores | <input checked="" type="checkbox"/> Difícil (volver al source) | <input checked="" type="checkbox"/> Fácil (ya está en staging) |
| Uso más común        | <input checked="" type="checkbox"/> 80% casos                  | <input checked="" type="checkbox"/> 20% casos                  |

### Cuándo usar cada tipo

#### Usar TEMPORARY cuando:

- Sources son estables y confiables
- Transformaciones son simples y robustas
- Espacio es limitado
- Velocidad es crítica
- Caso típico: cargas diarias estándar

#### Usar PERSISTENT cuando:

- Sources son inestables o cambian frecuentemente
- Transformaciones complejas (alto riesgo de error)
- Necesitas auditoría completa
- Regulaciones requieren mantener datos raw
- Caso típico: finanzas, salud, auditorías

## 2.2.5 Ejemplo Código SQL

### 2.2.5.1: Temporary Staging (ciclo diario):



```
-- DÍA 1
-- Paso 1: Truncar staging
TRUNCATE TABLE staging.sales;

-- Paso 2: Extract (full load primera vez)
INSERT INTO staging.sales
SELECT * FROM source_system.sales;

-- Paso 3: Transform y load a Core
INSERT INTO core.fact_sales (id, date, product, customer_id, store_id)
SELECT
 s.id,
 s.date,
 s.product,
 c.customer_id,
 st.store_id
FROM staging.sales s
JOIN dim_customers c ON s.customer_id = c.id
JOIN dim_stores st ON s.store_id = st.id;

-- Paso 4: Truncar staging
TRUNCATE TABLE staging.sales;

-- DÍA 2
-- Paso 1: Truncar staging
TRUNCATE TABLE staging.sales;

-- Paso 2: Extract (solo nuevos con delta logic)
INSERT INTO staging.sales
SELECT * FROM source_system.sales
WHERE date > '2022-01-02'; -- Delta logic por fecha

-- Paso 3: Transform y APPEND a Core
INSERT INTO core.fact_sales (id, date, product, customer_id, store_id)
SELECT
 s.id,
 s.date,
 s.product,
 c.customer_id,
 st.store_id
FROM staging.sales s
JOIN dim_customers c ON s.customer_id = c.id
JOIN dim_stores st ON s.store_id = st.id;

-- Paso 4: Truncar staging
TRUNCATE TABLE staging.sales;
```

## 2.2.5.2: Persistent Staging (ciclo diario):



```
-- DÍA 1
-- Paso 1: Extract (full load primera vez)
INSERT INTO staging.sales
SELECT * FROM source_system.sales;

-- Paso 2: Transform y load a Core
INSERT INTO core.fact_sales (id, date, product, customer_id, store_id)
SELECT
 s.id,
 s.date,
 s.product,
 c.customer_id,
 st.store_id
FROM staging.sales s
JOIN dim_customers c ON s.customer_id = c.id
JOIN dim_stores st ON s.store_id = st.id;

-- NO truncar

-- DÍA 2
-- Paso 1: Extract solo nuevos (delta logic)
INSERT INTO staging.sales
SELECT * FROM source_system.sales
WHERE date > '2022-01-02'; -- Delta logic

-- Paso 2: Transform solo nuevos
INSERT INTO core.fact_sales (id, date, product, customer_id, store_id)
SELECT
 s.id,
 s.date,
 s.product,
 c.customer_id,
 st.store_id
FROM staging.sales s
WHERE s.date > '2022-01-02' -- Solo procesar nuevos
JOIN dim_customers c ON s.customer_id = c.id
JOIN dim_stores st ON s.store_id = st.id;

-- NO truncar - staging ahora tiene datos de DÍA 1 + DÍA 2
```

## 2.2.6 Delta Logic: Columnas Delta + Conceptos clave

### ¿Cómo saber qué datos son nuevos?

Necesitas una columna delta que identifica datos nuevos:

#### Opción 1: ID autoincremental

sql

```
WHERE id > (SELECT MAX(id) FROM core.fact_sales)
```

**Problema:** IDs pueden no ser estrictamente incrementales.

#### Opción 2: Fecha/Timestamp (MÁS RECOMENDADO)

sql

```
WHERE created_at > (SELECT MAX(created_at) FROM core.fact_sales)
-- o
WHERE updated_at > (SELECT MAX(load_date) FROM core.fact_sales)
```

**Ventaja:** Fechas son más confiables.

#### Opción 3: Flag de procesado

sql

```
WHERE is_processed = FALSE
```



**Ventaja:** Control explícito de qué se ha cargado.

### Resumen Conceptos Clave

#### Staging Layer:

- Zona de aterrizaje para datos RAW
- Convierte TODO a tablas SQL
- Minimiza tiempo en systems fuente

#### Temporary Staging:

- Se trunca cada ciclo
- Solo datos actuales
- Más rápido, menos espacio

#### Persistent Staging:

- Nunca se trunca
- Histórico completo
- Backup permanente

#### Delta Logic:

- Identifica datos nuevos
- Columnas: ID, fecha, flag
- Evita reprocesar datos viejos

## 2.2.7 Sobre Columna Delta

Columna que identifica QUÉ datos son NUEVOS desde la última carga.

---

### Ejemplo:

Día 1 cargaste hasta:

```
id=5, fecha=2022-01-02
```

Día 2 necesitas cargar solo lo nuevo:

```
sql

-- Columna delta = "id"
SELECT * FROM source WHERE id > 5

-- O columna delta = "date"
SELECT * FROM source WHERE date > '2022-01-02'
```

Resumen:

**Columna delta = Columna que te dice "desde aquí para adelante es nuevo"**

Típicamente:

- id (autoincremental)
- created\_at (fecha creación)
- updated\_at (fecha modificación)

## 2.2.8 Schema vs Tabla + Analogía

### SCHEMA (Esquema)

= Carpeta/Contenedor que agrupa tablas

```
Database: DataWarehouseX
 └── Schema: staging
 ├── tabla: sales
 ├── tabla: customers
 └── tabla: products

 └── Schema: core
 ├── tabla: dim_customers
 ├── tabla: fact_sales
 └── tabla: dim_products
```

**Propósito:** Organizar y separar capas del DWH

### TABLA (Table)

= Archivo que contiene los datos reales

```
Schema: staging
└── Tabla: sales
 id | date | product
 1 | 2022-01-02 | Tomato
 2 | 2022-01-02 | Chilli
```

### Analogía Simple

```
Database = Disco duro
└── Schema = Carpeta
 └── Tabla = Archivo Excel
```

## 2.2.8.1 Código SQL

Crear Schema:

```
sql
CREATE SCHEMA staging;
```

Crear Tabla dentro del Schema:

```
sql

CREATE TABLE staging.sales (
 id INT,
 date DATE,
 product VARCHAR(100)
);

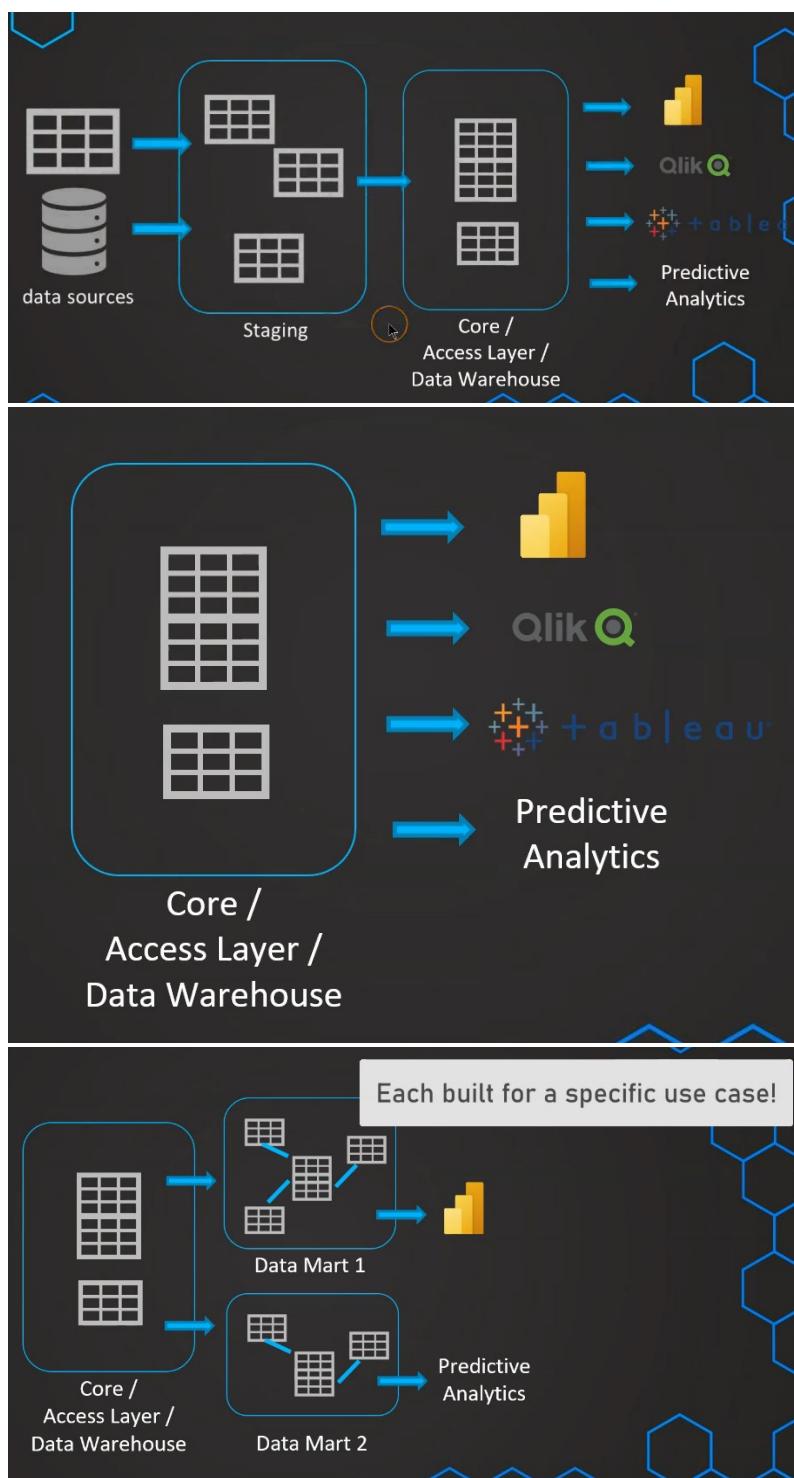
🌟 Resumen Ultra Breve

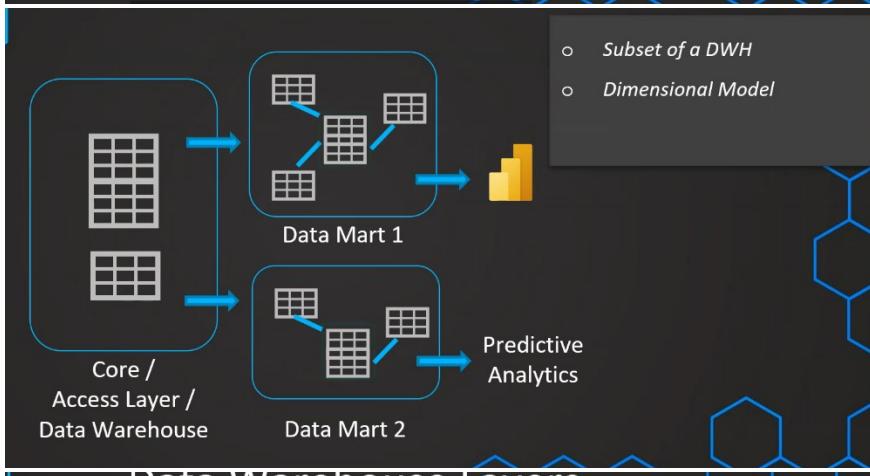
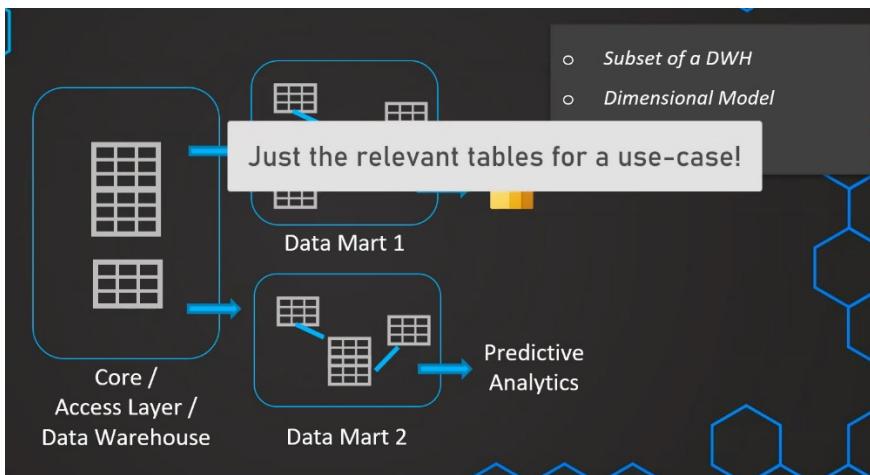
- **Schema** = Carpeta (organiza tablas por capa)
- **Tabla** = Archivo (guarda los datos)

staging (schema)
└─ sales (tabla con datos)
 └─ products (tabla con datos)

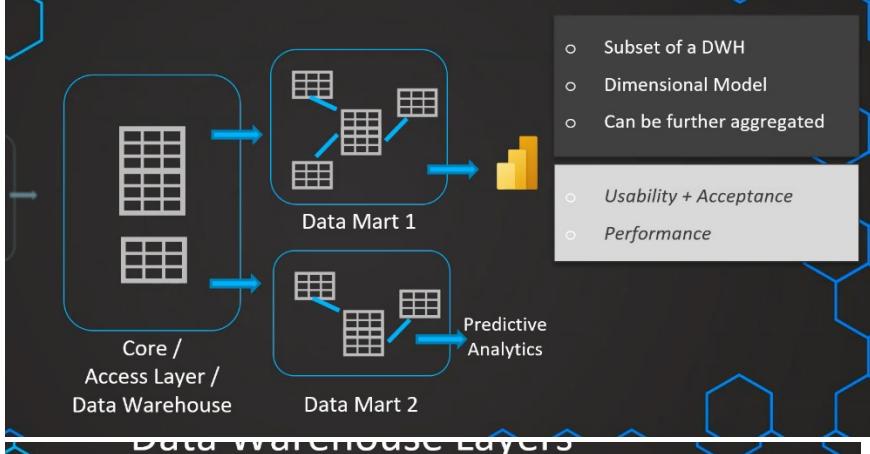
core (schema)
└─ fact_sales (tabla con datos)
```

## 2.3: Data Marts

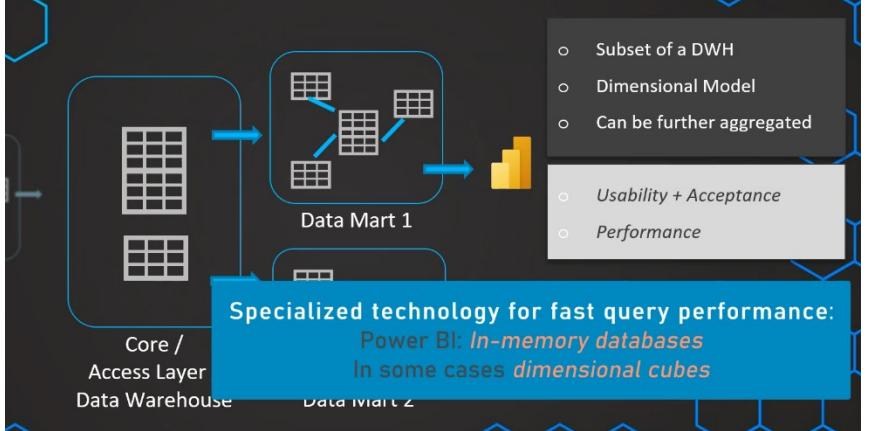


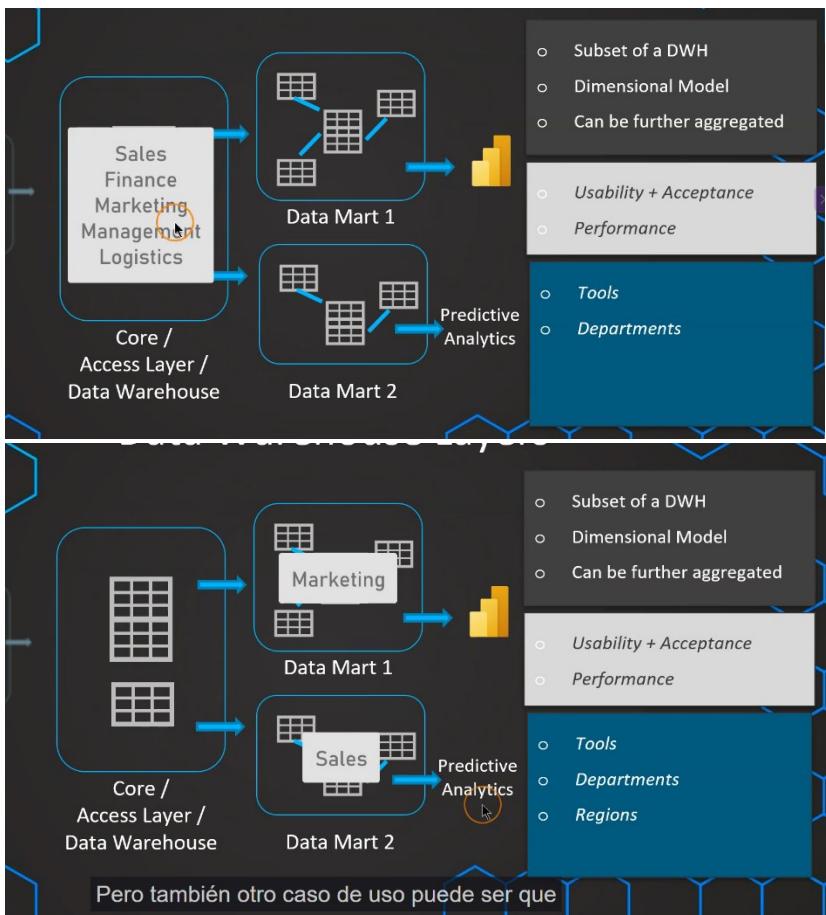


## Data Warehouse Layers

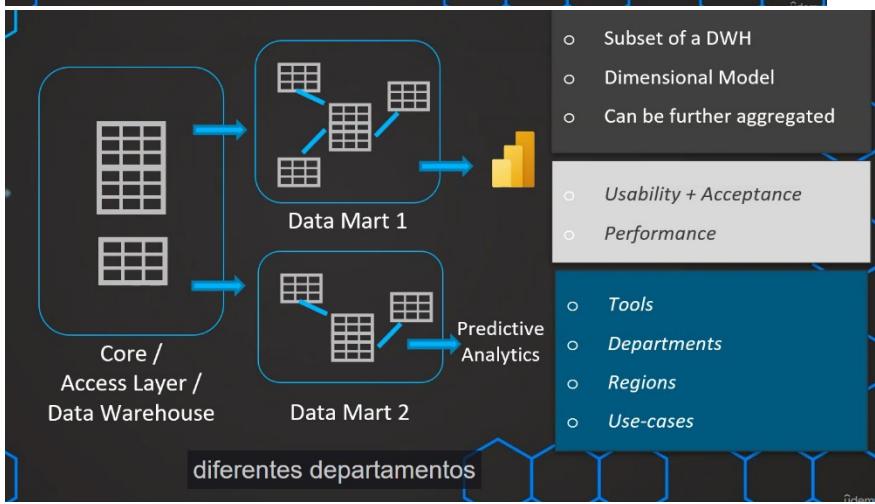


## Data Warehouse Layers





Pero también otro caso de uso puede ser que



diferentes departamentos

✓ **Data Mart = Small scale DWH?**

⇒ **Focus on the business problem**

✓ **Should you use a Data Mart or not?**

⇒ **Focus on the business problem**

### **2.3.1 ¿Qué es un Data Mart?**

**Data Mart = Subconjunto especializado del Data Warehouse Core para un caso de uso específico.**

**Analogía:** Si el Data Warehouse Core es un supermercado completo, un Data Mart es una sección especializada (panadería, carnicería, verdulería).

### **2.3.2 Arquitectura con Data Marts**

#### **Opción 1: Sin Data Marts (más simple)**

```
Sources → Staging → Core/DWH → Power BI
 → Tableau
 → Predictive Analytics
```

**Core sirve directamente a TODOS los usuarios.**

#### **Opción 2: Con Data Marts (empresas grandes)**

```
Sources → Staging → Core/DWH → Data Mart 1 (Marketing) → Power BI
 → Data Mart 2 (Sales) → Predictive Analytics
 → Data Mart 3 (Finance) → Tableau
```

**Core alimenta Data Marts especializados.**

### **2.3.3 Características de un Data Mart**

#### **1. Subconjunto del DWH**

**Solo contiene tablas relevantes para un caso de uso específico.**

Core DWH (150 tablas):

- Sales (50 tablas)
- Finance (40 tablas)
- Marketing (30 tablas)
- Logistics (30 tablas)

↓ Crear Data Mart ↓

Data Mart Marketing (solo 30 tablas):

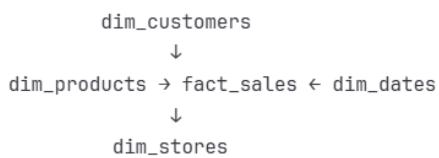
- dim\_customers
- dim\_campaigns
- fact\_conversions
- dim\_channels

## 2. Modelo Dimensional

Datos organizados en:

- **Fact Tables** (hechos): Métricas, números (ventas, conversiones)
- **Dimension Tables** (dimensiones): Contexto (clientes, productos, tiempo)

Estructura Star Schema:



## 3. Puede estar agregado

Datos pre-calculados para velocidad.

Core DWH:

- Transacciones individuales (millones de filas)

Data Mart:

- Ventas agrupadas por mes/producto (miles de filas)
- Más rápido para dashboards

## 2.3.4 Ventajas de usar Data Marts

### 2.3.4.1: 1. Usabilidad + Aceptación

Problema sin Data Mart:

Usuario de Marketing abre DWH:  
"¿Cuál de estas 150 tablas necesito? 🤯"

Solución con Data Mart:

Usuario de Marketing abre Data Mart Marketing:  
"Solo 20 tablas relevantes, ¡perfecto! 😊"

**Beneficio:** Usuarios NO técnicos no se sienten abrumados.

### 2.3.4.2: 2. Performance (Rendimiento)

Tecnologías especializadas para velocidad:

- **In-memory databases:** Datos en RAM (ultra rápido)
- **OLAP Cubes:** Pre-calculados para consultas complejas
- **Índices optimizados:** Para consultas específicas

Ejemplo:

Core DWH (PostgreSQL estándar):

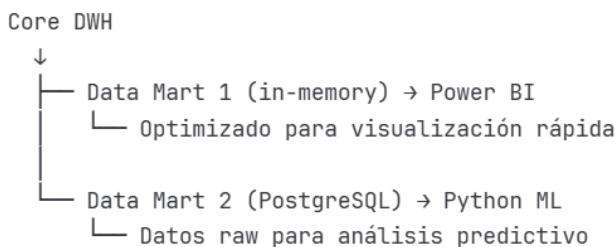
- Query tarda 30 segundos

Data Mart (in-memory):

- Misma query tarda 2 segundos

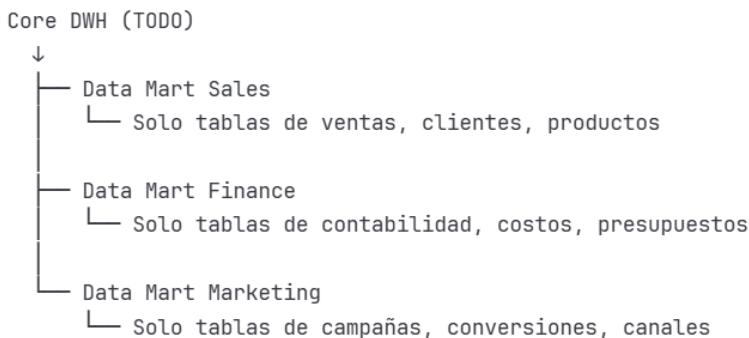
## 2.3.5 Casos de Uso: Cuándo crear Data Marts

### Caso 1: Diferentes Herramientas



**Por qué:** Cada herramienta tiene necesidades diferentes.

### Caso 2: Diferentes Departamentos



**Ejemplo concreto:**

**Core DWH contiene:**

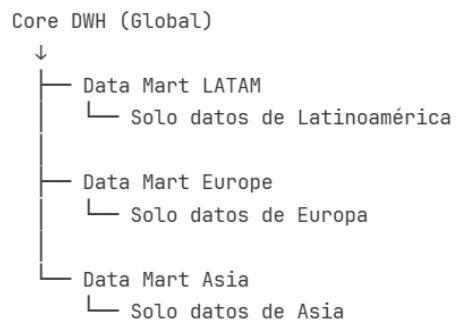
- Sales (50 tablas)
- Finance (40 tablas)
- Marketing (30 tablas)
- Logistics (30 tablas)
- **Total: 150 tablas**

**Data Mart Marketing contiene:**

- dim\_customers
- dim\_campaigns
- dim\_channels
- fact\_conversions
- fact\_ad\_spend
- **Total: 30 tablas (solo relevantes para Marketing)**

**Ventaja:** Equipo Marketing NO ve las 120 tablas irrelevantes.

### Caso 3: Diferentes Regiones



**Por qué:** Cada región solo necesita sus propios datos.

## 2.3.6 Ejemplo Practico Completo

Empresa: E-commerce Global

Situación:

- 5 departamentos (Sales, Marketing, Finance, Logistics, HR)
- regiones (LATAM, Europe, Asia)
- herramientas BI (Power BI, Tableau)

---

Arquitectura SIN Data Marts:

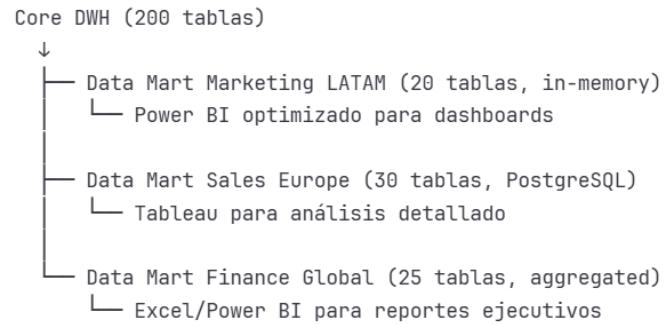


Problemas:

- ✗ Usuarios abrumados con 200 tablas
- ✗ Consultas lentas (todos consultan el Core)
- ✗ No optimizado para cada caso de uso

---

Arquitectura CON Data Marts:



Beneficios:

- Cada equipo solo ve tablas relevantes
- Consultas más rápidas (menos datos)
- Tecnología optimizada por caso de uso

## 2.3.7 Proceso de Creación de Data Mart

### Paso 1: Identificar caso de uso

¿Quién lo usará? → Equipo Marketing

¿Para qué? → Dashboards de campañas

¿Qué herramienta? → Power BI

### Paso 2: Seleccionar tablas relevantes del Core

```
sql
-- Del Core DWH, copiar solo tablas relevantes
CREATE TABLE dm_marketing.dim_customers AS
SELECT * FROM core.dim_customers;

CREATE TABLE dm_marketing.dim_campaigns AS
SELECT * FROM core.dim_campaigns;

CREATE TABLE dm_marketing.fact_conversions AS
SELECT * FROM core.fact_conversions;
```

### Paso 3: Agregar/optimizar datos (opcional)

```
sql
-- Pre-calcular métricas para velocidad
CREATE TABLE dm_marketing.monthly_conversions AS
SELECT
 DATE_TRUNC('month', conversion_date) AS month,
 campaign_id,
 COUNT(*) AS total_conversions,
 SUM(revenue) AS total_revenue
FROM core.fact_conversions
GROUP BY month, campaign_id;
```

### Paso 4: Implementar en tecnología especializada

Opción A: In-memory (para Power BI)

- Cargar Data Mart en Redis/Memcached
- Consultas en milisegundos

Opción B: OLAP Cube

- Pre-calcular todas las agregaciones posibles
- Drill-down super rápido

### 2.3.8 Core DWH vs Data Mart + Sobre la terminología

| Aspecto         | Core DWH          | Data Mart              |
|-----------------|-------------------|------------------------|
| Contenido       | TODO              | Subconjunto específico |
| Tablas          | 100-500+          | 10-50                  |
| Usuarios        | Todos (potencial) | Equipo específico      |
| Propósito       | Centralizado      | Caso de uso específico |
| Performance     | Estándar          | Optimizado             |
| Tecnología      | PostgreSQL/MySQL  | In-memory, OLAP cubes  |
| Actualizaciones | Desde Staging     | Desde Core             |

#### Importante: NO obsesionarse con terminología

El instructor enfatiza:

"No hay que obsesionarse demasiado con la terminología, sino centrarse en el problema empresarial."

Debate común:

#### Pregunta 1: ¿Data Mart = Data Warehouse pequeño?

Respuestas:

- Algunos dicen: "Sí, básicamente"
- Otros dicen: "¡No! Es incorrecto"

La realidad pragmática:

Si llamarlo "Data Warehouse pequeño" te ayuda a entenderlo  
→ Está bien  
→ Lo importante es resolver el problema de negocio

#### Pregunta 2: ¿Usar Data Mart o solo DWH?

Respuesta práctica:

Depende del problema de negocio:

¿Tienes 1 equipo y 20 tablas?  
→ NO necesitas Data Marts

¿Tienes 5 equipos, 3 regiones y 200 tablas?  
→ SÍ necesitas Data Marts

## 2.3.9 Reglas Prácticas: Cuándo usar Data Marts + Ejemplo Visual Completo + Puntos Clave para Recordar

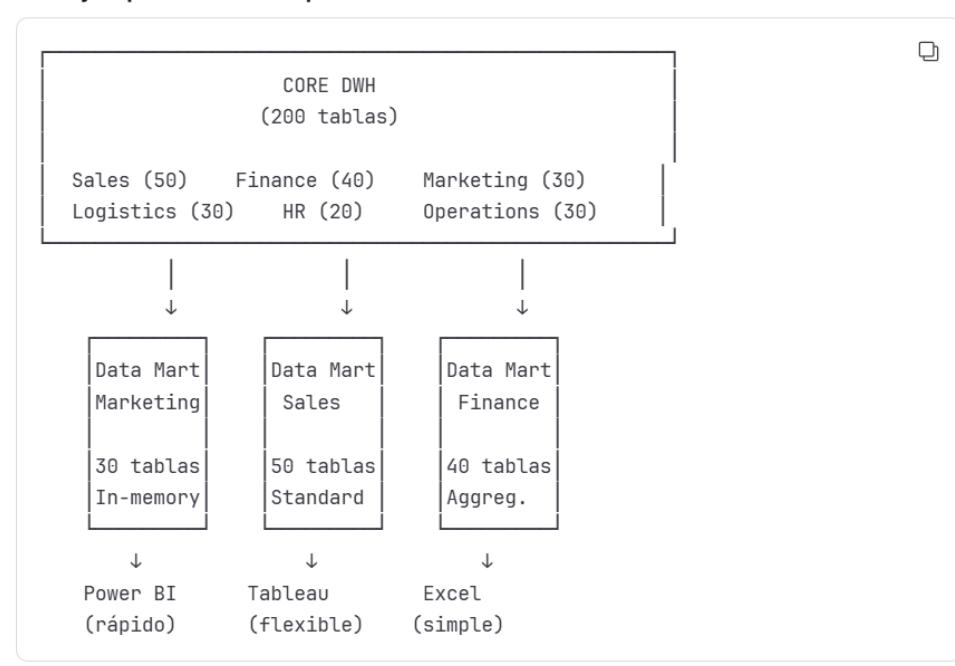
Usa Data Marts cuando:

- DWH tiene 100+ tablas
- Múltiples departamentos/equipos
- Diferentes herramientas BI
- Necesitas optimización de performance
- Usuarios NO técnicos se sienten abrumados

NO uses Data Marts cuando:

- DWH tiene <50 tablas
- Un solo equipo pequeño
- Una sola herramienta BI
- Performance es aceptable
- Usuarios pueden navegar el Core fácilmente

## Ejemplo Visual Completo



## Puntos Clave para Recordar

**1. Data Mart = Subconjunto especializado del Core DWH**

**2. razones principales para usar:**

- Diferentes herramientas
- Diferentes departamentos
- Diferentes regiones

**3. ventajas principales:**

- Usabilidad (menos tablas, más fácil)
- Performance (tecnología optimizada)

**4. Pragmatismo sobre terminología:**

- Enfocarse en resolver problemas de negocio
- No debatir definiciones académicas

**5. Decisión basada en necesidad:**

- Empresa pequeña → Probablemente NO necesitas
- Empresa grande → Probablemente SÍ necesitas

## **2.3.10: Resumen Visual + Conclusión Práctica**

SIN Data Marts (simple):

Sources → Staging → Core → Power BI/Tableau/etc



CON Data Marts (complejo):

Sources → Staging → Core → Data Mart 1 → Power BI  
→ Data Mart 2 → Tableau  
→ Data Mart 3 → Predictive

¿Cuándo usar cada uno?

- DEPENDE del problema de negocio
- No hay respuesta única
- Evalúa tu caso específico

## **Conclusión Práctica**

**Data Marts son OPCIONALES.**

**Úsalos cuando:**

- Mejoren usabilidad para usuarios
- Mejoren performance significativamente
- Resuelvan problema de negocio real

**NO los uses:**

- "Porque sí"
- "Porque está en el libro"
- "Porque es la arquitectura correcta"

**Enfócate en:** ¿Esto resuelve MI problema de negocio?

## 2.3.11: + In-Memory vs OLAP Cubes

### 2.3.11.1: In-Memory Database

Qué es: Base de datos que guarda TODO en **RAM** (no en disco).

Cómo funciona:

Disco: 100ms para leer  
RAM: 0.1ms para leer (1000x más rápido)

Tecnologías:

- Redis
- SAP HANA
- MemSQL
- VoltDB

### 2.3.11.2: OLAP Cubes

Qué es: Datos **pre-calculados** en todas las combinaciones posibles.

Cómo funciona:

Pregunta: "Ventas por mes, producto, región"

Sin Cube: Calcular ahora (30 seg)

Con Cube: Ya está calculado (0.5 seg)

Tecnologías:

- Microsoft SSAS (SQL Server Analysis Services)
- Oracle OLAP
- IBM Cognos
- Mondrian

Uso: Reportes con drill-down (expandir/colapsar dimensiones).

### 2.3.11.3: Diferencia Principal + Core con Dimensiones: SÍ, PUEDE

#### VS Diferencia Principal

| Aspecto      | In-Memory          | OLAP Cube                  |
|--------------|--------------------|----------------------------|
| Estrategia   | Datos en RAM       | Pre-calcular TODO          |
| Velocidad    | Por hardware (RAM) | Por pre-cálculo            |
| Espacio      | Igual que original | 10-100x más grande         |
| Flexibilidad | Consultas libres   | Solo dimensiones definidas |

#### **2.3.11.4 Core con Dimensiones: SÍ, PUEDE + Resumen**

El Core PUEDE tener modelo dimensional (Star Schema):

Core DWH:

```
└── dim_customers ← Dimensión
└── dim_products ← Dimensión
└── dim_dates ← Dimensión
└── fact_sales ← Hechos
```

**Cuándo:**

- Si Core sirve directamente a usuarios (sin Data Marts)
- Si ya está optimizado para BI

**Cuando NO:**

- Core solo almacena (tablas normalizadas)
- Data Marts hacen el modelado dimensional

---

**Resumen:**

- **In-memory:** RAM = velocidad
- **OLAP Cubes:** Pre-cálculo = velocidad
- **Core con dimensiones:** Sí, totalmente válido

### **2.3.12: + In-Memory vs OLAP Cubes v2**

NO exactamente.

**In-Memory:**

- Calcula **en tiempo real** (pero ultra rápido porque usa RAM)
- NO pre-calcula nada
- Tamaño: Lo que quepa en RAM (puede ser grande si tienes mucha RAM)

**OLAP Cube:**

- **NO calcula**, ya está todo pre-calculado
- Solo lee resultados guardados
- Tamaño: Cualquiera (se guarda en disco)

---

#### **Diferencia Clave:**

In-Memory:

Usuario pregunta → Calcula AHORA (0.1 seg) → Respuesta

OLAP Cube:

Usuario pregunta → Lee pre-calculado (0.5 seg) → Respuesta

---

#### **💡 Resumen:**

**In-Memory = Calcular rápido (RAM)**

**OLAP Cube = NO calcular (ya está hecho)**

La diferencia NO es tamaño, es **estrategia de velocidad**.

**2.3.12.1 cual es el sentido de tener olap cube si in memory lo hace mas rapido y aparte que es libre?**

### **2.3.12.1: 1. Límite de RAM (problema físico)**

Datos: 500GB  
RAM disponible: 64GB  
In-Memory:  NO cabe  
OLAP Cube:  Se guarda en disco

#### **Costo:**

- 500GB de disco: ~\$10
- 500GB de RAM: ~\$5,000

---

### **2.3.12.2: 2. Drill-Down complejo**

**Pregunta complicada:** "Ventas por año → mes → día → hora → producto → categoría → región → ciudad"

**In-Memory:**

sql □  
-- Tienes que calcular TODAS estas agregaciones:  
GROUP BY year  
GROUP BY year, month  
GROUP BY year, month, day  
GROUP BY year, month, day, hour  
...  
(8 niveles = 8 queries diferentes)  
```

OLAP Cube:

Ya tiene TODAS las combinaciones guardadas Click en año → mes → día (instantáneo)

2.3.12.3: 3 Persistencia

In-Memory: Server se apaga → Datos desaparecen (en RAM)

Hay que recargar todo

OLAP Cube: Server se apaga → Datos siguen ahí (en disco)

2.3.12.2: Cuando usar cada uno + Respuesta Directa

In-Memory (mejor cuando):

- Datos caben en RAM (<100GB típicamente)
- Queries simples (SELECT, WHERE, GROUP BY básicos)
- Dashboards en tiempo real
- Tienes presupuesto para RAM

OLAP Cube (mejor cuando):

- Datos enormes (500GB - 10TB)
- Análisis dimensional complicado (drill-down de 5+ niveles)
- Presupuesto limitado (disco es barato)
- Necesitas persistencia

 **Respuesta directa:**

"**¿Por qué usar OLAP si In-Memory es más rápido?**"

Porque In-Memory:

-  Es MUY caro (RAM cara)
-  Tiene límite físico (RAM limitada)
-  No persiste (se pierde al apagar)

Para datos pequeños (<100GB): In-Memory gana

Para datos grandes (>500GB): OLAP Cube gana

2.3.12.3: ¿Power bi es inmemory?

 **Respuesta: Sí (mayormente)**

Power BI usa tecnología In-Memory llamada "VertiPaq"

Cómo funciona:

1. Importas datos a Power BI
↓
2. Power BI los carga en RAM (comprimidos)
↓
3. Visualizaciones ultra rápidas (todo en memoria)

Compresión: 10GB de datos → 1GB en RAM (compresión columnar)

3 Modos de Power BI:

1. Import Mode (In-Memory) ★ Más común

- Datos en RAM
- Ultra rápido
- Límite: Lo que quepa en RAM

2. DirectQuery (NO In-Memory)

- Sin límite de tamaño
- Datos siempre actualizados
- Más lento (consulta la BD cada vez)

3. Live Connection

Conecta a SSAS o Azure Analysis Services
(depende de dónde estén los datos)

❖ Resumen:

Power BI = Principalmente In-Memory

Por eso es tan rápido, pero tiene límites de RAM.

2.3.12.4 OLAP Cubes más populares – Microsoft SASS – Oracle OLAP – IBM Cognos - Mondrian

Recordatorio de:

Databricks y Spark son para OTRA cosa:

- Procesamiento Big Data
- ETL/ELT masivo
- Data Lakes
- **NO son OLAP Cubes**

1. Microsoft SSAS ★ MÁS POPULAR

- Líder del mercado
- Se integra con Power BI
- Enterprise (Windows)

2. Oracle OLAP

- Para empresas grandes con Oracle
- Enterprise

3. IBM Cognos

- Para empresas IBM
- Enterprise

4. Mondrian (Open Source)

- Gratis
- Java-based

2.3.12.4.1: En la practica (2024) + Resumen

Power BI ya reemplazó a muchos OLAP Cubes tradicionales

Antes:

DWH → SSAS Cube → Excel/Reportes

Ahora:

DWH → Power BI (in-memory) → Dashboards

💡 Resumen:

OLAP Cubes: SSAS (Microsoft) es el rey

Databricks/Spark: Para Big Data/ETL, NO para OLAP Cubes

2.3.12.4.2: Aclaración de Power BI que reemplazo a OLAP Cubes

Power BI reemplazó OLAP Cubes para empresas PEQUEÑAS/MEDIANAS, pero NO para ENTERPRISE.

Comparación Real:

Power BI (Import Mode)

Límite: ~10GB comprimido (~100GB datos raw)

Velocidad: Ultra rápido

Costo: ~\$10/usuario/mes

Complejidad: Baja (fácil de usar)

Ideal para: 80% de empresas (pequeñas/medianas)

SSAS Cube

Límite: Sin límite (TB de datos)

Velocidad: Rápido

Costo: \$\$\$ (licencias SQL Server Enterprise)

Complejidad: Alta (necesitas DBA/especialistas)

Ideal para: Empresas grandes con TB de datos

2.3.12.4.3: Arquitectura Moderna + Elección + Resumen

Empresa Pequeña/Mediana:

- DWH (100GB) → Power BI (in-memory) → Dashboards
- Simple, barato, suficiente

Empresa Grande (Enterprise):

- DWH (5TB) → SSAS Cube (pre-calculado) → Power BI (conecta al cube) → Dashboards
- Power BI NO puede cargar 5TB en RAM
 - SSAS Cube maneja los TB de datos
 - Power BI solo visualiza

¿Elegir entre SSAS Cube y Power BI?

- Datos < 100GB → Power BI solo
- Datos > 500GB → SSAS Cube + Power BI

❖ Resumen:

Power BI reemplazó cubes tradicionales para casos PEQUEÑOS
SSAS Cubes siguen vivos para ENTERPRISE con TB de datos

No es "uno u otro", muchas veces se usan JUNTOS.

2.3.12.4.4: SSAS Cube = Consultas pre-calculadas súper rápidas a ESCALA MASIVA

Lo Especial de SSAS Cube:

1. Drill-Down Instantáneo (Interfaz especial)

```
sql

-- Query 1: Ver por año
SELECT year, SUM(sales) GROUP BY year;

-- Query 2: Ver por mes (tienes que escribir otra query)
SELECT year, month, SUM(sales) GROUP BY year, month;

-- Query 3: Ver por día (otra query más)
SELECT year, month, day, SUM(sales) GROUP BY year, month, day;
```

Con SSAS Cube:

```
Click en 2024 → Automáticamente expande a meses
Click en Enero → Automáticamente expande a días
Click en día 15 → Automáticamente expande a horas

(TODO pre-calculado, 0 queries escritas)
```

2. Cálculos Complejos Pre-definidos

Ejemplo: % de variación año anterior

Sin Cube:

```
sql
-- Query SÚPER compleja
SELECT
    year,
    sales,
    LAG(sales) OVER (ORDER BY year) as prev_year,
    (sales - LAG(sales) OVER (ORDER BY year)) / LAG(sales) * 100 as pct_change
FROM ...
```

Con SSAS Cube:

```
Defines UNA VEZ: "YoY% = (Este Año - Año Anterior) / Año Anterior * 100"
Luego: Automático en TODOS los reportes
```

3. Lenguaje Especial: MDX

MDX permite consultas "raras" que SQL no hace fácil:



```
SELECT  
    TopCount([Product].[Product].Members, 10, [Measures].[Sales]) ON ROWS,  
    {[Time].[Last 3 Months], [Time].[Same Period Last Year]} ON COLUMNS  
FROM [Sales Cube]
```

En SQL esto serían 20+ líneas complicadas.

😮 Entonces... ¿Vale la pena?

Honestamente:

Para la MAYORÍA de empresas: NO

...

Power BI hace el 90% del trabajo

Más fácil, más barato

...

Para ENTERPRISE con:

- TB de datos
- 1000+ usuarios concurrentes
- Análisis complejos (drill-down de 10 niveles)
- Cálculos financieros sofisticados

...

SSAS Cube: SÍ vale la pena

💡 Respuesta Honesta:

Sí, SSAS Cube básicamente es "consultas pre-calculadas rápidas".

La diferencia es:

- ESCALA (TB de datos, miles de usuarios)
- COMPLEJIDAD (drill-down automático, MDX, cálculos complejos)
- INTERFAZ (no escribes queries, solo haces click)

Para tu caso (freelance, proyectos medianos): Power BI es suficiente.

Para bancos, corporaciones, gobierno: SSAS Cube tiene sentido.

2.3.12.4.4: ¿SSAS Cube sirve para BI? + Arquitectura correcta + Ejemplo real

- 🎯 SSAS Cube: SOLO para BI
- ✗ NO se usa para ML/Data Science

vs Por qué NO sirve para ML:

SSAS Cube tiene datos AGREGADOS:

Datos originales (para ML):

customer_id	purchase_date	product	amount
1	2024-01-15	Laptop	\$1000
1	2024-01-20	Mouse	\$20
2	2024-01-16	Laptop	\$1000

Datos en SSAS Cube (agregados):

month	product	total_sales	avg_amount
2024-01	Laptop	\$2000	\$1000
2024-01	Mouse	\$20	\$20

Problema para ML:

ML necesita: Cada transacción individual
SSAS Cube tiene: Solo totales/promedios



- ✗ Perdiste el detalle
- ✗ No puedes entrenar modelos

Arquitectura Correcta:

Para BI (SSAS Cube):

Data Warehouse → SSAS Cube → Power BI/Reportes

- Datos agregados
- Pre-calculados
- Rápido para dashboards

Para ML (Data Lake):

Data Lake (raw data) → Spark/Python → ML Models

- Datos individuales
- Sin agregar
- Todos los detalles disponibles

Ejemplo Real:

Banco grande con ambos:

Data Lake (Raw)
- Cada transacción individual
- Clicks en app
- Logs de ATM



ML
Models

DWH + SSAS
Cube



Detección
Fraude



Dashboards
Ejecutivos

Separados y diferentes:

- Data Lake → ML: Datos raw completos
- DWH + SSAS → BI: Datos agregados rápidos

2.3.12.5: Databricks se usa para ML en Data Lake, PERO también para MUCHO más. +

Resumen

Usos de Databricks:

1. ETL/ELT (procesamiento datos) Uso principal

Data Lake (raw) → Databricks (Spark) → Transforma → Data Warehouse

Ejemplo: Limpiar 500GB de logs diarios

2. ML/Data Science

Data Lake (raw) → Databricks (notebooks Python/R) → Entrenar modelos

Ejemplo: Modelo de recomendación con millones de filas

3. Análisis Exploratorio

Data Lake (raw) → Databricks (SQL/Python) → Insights

Stack Típico Data Lake:

Sources → Data Lake (S3/ADLS) → Databricks (procesar TODO) → DWH
↓
ML Models

Databricks = Cerebro del Data Lake

- Procesa Big Data (Spark)
- Hace ETL
- Entrena ML
- Analiza datos

Resumen:

Databricks: ETL/ELT (uso más común)

- ML/Data Science
- Análisis exploratorio
- Procesamiento Big Data

NO es solo para ML, es para TODO lo que hagas con Data Lake.

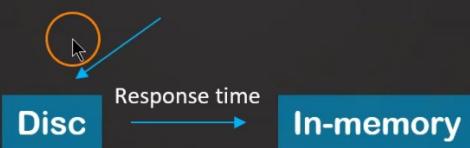
2.4: In-Memory Databases (Bases de datos en Memoria)

In-memory databases

- ✓ Highly optimized for query performance
- ✓ Good for Analytics / High query volume
- ✓ Usually used for data marts
- ✓ Relational and non-relational



Traditional database



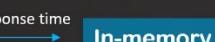
Traditional database



In-memory database



Disc



In-memory



Traditional database



In-memory database

- columnar storage,
- parallel query plans,
- and other techniques



Disc



In-memory

- *Durability*: Lose all information when device loses power or is reset
- Durability added through snapshots / images
- Cost-factor
- Traditional DBs also trying reduce usage of disc

✓ SAP HANA

✓ MS SQL Server In-Memory Tables

✓ Oracle In-Memory

✓ Amazon MemoryDB

2.4.1 ¿Qué es una In-Memory Database? + Problema que resuelve + Como Funciona

Base de datos que almacena **TODOS** los datos en RAM (memoria) en lugar de disco.

Ventaja: Elimina el tiempo de leer desde el disco → Ultra rápido

Desventaja: Más caro, riesgo de pérdida de datos

Problema que Resuelve

Traditional Database (lenta):

```
Usuario hace query  
↓  
BD busca datos en DISCO 🏟 (100ms)  
↓  
Carga datos en RAM  
↓  
Procesa query  
↓  
Resultado (LENTO - mayoría del tiempo en leer disco)
```

Tiempo total: 100-500ms (dependiendo del disco)

In-Memory Database (rápida):

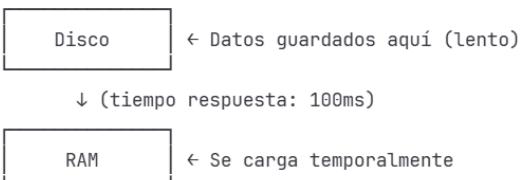
```
Usuario hace query  
↓  
BD busca datos en RAM ↗ (0.1ms)  
↓  
Procesa query  
↓  
Resultado (RÁPIDO - sin leer disco)
```

Tiempo total: 1-10ms (1000x más rápido)

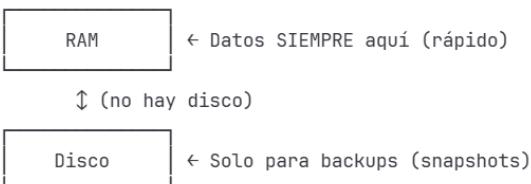
Cómo Funciona

1. Almacenamiento: TODO en RAM

Traditional DB:



In-Memory DB:



2. Técnicas Adicionales de Velocidad

Traditional (Row-based):

Query grande: "Analizar ventas de 10 millones de filas"

Proceso: 1 thread → procesa todo secuencialmente → 60 segundos

In-Memory (Column-based):

Query grande: "Analizar ventas de 10 millones de filas" 

Proceso:

- Thread 1 → procesa filas 1-2.5M → 15 seg
- Thread 2 → procesa filas 2.5M-5M → 15 seg
- Thread 3 → procesa filas 5M-7.5M → 15 seg
- Thread 4 → procesa filas 7.5M-10M → 15 seg

Total: 15 segundos (4x más rápido)

Ventaja: 10-100x más rápido para análisis (solo lee columnas necesarias).

Parallel Query Plans (Planes de Consulta en Paralelo)

Traditional:

Query grande: "Analizar ventas de 10 millones de filas"

Proceso: 1 thread → procesa todo secuencialmente → 60 segundos

In-Memory:

Query grande: "Analizar ventas de 10 millones de filas"

Proceso:

- Thread 1 → procesa filas 1-2.5M → 15 seg
- Thread 2 → procesa filas 2.5M-5M → 15 seg
- Thread 3 → procesa filas 5M-7.5M → 15 seg
- Thread 4 → procesa filas 7.5M-10M → 15 seg

Total: 15 segundos (4x más rápido)

2.4.2 Casos de Uso + No ideal para

Ideal para:

Data Marts (capa de acceso final)

Core DWH (disco) → Data Mart (in-memory) → Power BI/Tableau
Usuarios no esperan → Dashboards instantáneos

Analítica en tiempo real

Dashboard CEO: Ventas actualizadas cada segundo



Alto volumen de consultas

1000 usuarios consultando simultáneamente

Consultas complejas

Query: "Ventas por producto, región, mes, cliente, categoría (últimos 5 años)"

Traditional DB: 2 minutos

In-Memory DB: 2 segundos

NO ideal para:

Datos que no caben en RAM (>500GB típicamente) Presupuesto limitado Datos que cambian constantemente (OLTP) Baja frecuencia de consultas

2.4.3 Desafíos

2.4.3.1: 1. Durabilidad (Problema Principal)

⚠ IMPORTANTE: Hay 2 escenarios diferentes

Escenario A: In-Memory Database PURA (sin respaldo)

Problema:

Server con In-Memory DB (única copia de datos)

↓

Se va la luz ⚡*

↓

RAM se borra (datos volátiles)

↓

TODOS los datos perdidos 😱 (NO HAY RESPALDO)



Este es el problema real cuando NO hay otra copia.

Escenario B: Data Mart In-Memory (con Core DWH de respaldo)

SIN problema:

Data Mart In-Memory (copia en RAM)

↓

Se va la luz ⚡*

↓

RAM se borra (Data Mart desaparece)

↓

Core DWH en disco NO se afecta ✓

↓

Reiniciar y recargar desde Core DWH

↓

DATOS NO SE PIERDEN ✓ (hay respaldo en Core)

Este es el caso común en Data Warehousing.

Conclusión sobre durabilidad:

- Si In-Memory es tu ÚNICA copia: Necesitas snapshots/WAL (crítico)
 - Si In-Memory es una COPIA del Core DWH: Solo necesitas tiempo de recarga (no crítico)
-

Soluciones para In-Memory Database PURA (Escenario A):

Opción A: Snapshots/Imágenes

Cada X horas:

1. Pausar escrituras
2. Copiar RAM completa → Disco (snapshot)
3. Reanudar escrituras

Si se cae el server:

1. Reiniciar
2. Cargar último snapshot → RAM
3. Perder solo datos desde último snapshot

Ejemplo:

10:00 AM → Snapshot guardado

10:30 AM → Server se cae

Reinicio: Carga snapshot de 10:00 AM

Resultado: Perdiste 30 min de datos

Opción B: Write-Ahead Log (WAL)

Cada escritura:

1. Escribir cambio en disco (log)
2. Aplicar cambio en RAM

Si se cae el server:

1. Reiniciar
2. Cargar último snapshot
3. Aplicar logs pendientes

Resultado: 0 datos perdidos

Desventaja: Más lento (escribe en disco), pero más seguro.

2.4.3.2: 2. Costo

Comparación de costos:

Disco SSD: \$0.10 por GB
RAM: \$5-10 por GB (50-100x más caro)



Ejemplo: 1TB de datos

- En disco: \$100
- En RAM: \$5,000-10,000

Además:

RAM no es infinita:
- Server típico: 64-256GB RAM
- Para 1TB datos: Necesitas múltiples servers (más caro aún)

2.4.3.3: 3. Traditional DBs están mejorando

Innovaciones recientes:

- SSD NVMe ultra rápidos (reducen tiempo lectura disco)
- Caching inteligente (datos frecuentes en RAM automáticamente)
- Compresión avanzada

Resultado:

Traditional DB (2010): 500ms query
Traditional DB (2024): 50ms query (10x mejora)
In-Memory DB: 5ms query

Diferencia ya no es tan dramática

2.4.4: Ejemplos de In-Memory Databases + Comparación Completa

2.4.3.1: SAP HANA (Líder Enterprise)

Quién lo usa:

- Grandes corporaciones (SAP ERP)
- Bancos, retail, manufactura

Características:

- Columnar storage
- In-memory completo
- Integración con SAP ecosystem

Ejemplo:

Caso: Walmart

Antes (Traditional DB): Análisis inventario global → 1 hora

Después (SAP HANA): Análisis inventario global → 3 segundos

2.4.3.2: Microsoft SQL Server In-Memory Tables

Qué es:

- SQL Server tradicional + tablas in-memory opcionales

Ventaja:

```
sql  
-- Tabla tradicional (disco)  
CREATE TABLE sales (...);  
  
-- Tabla in-memory (RAM)  
CREATE TABLE sales_fast (...)  
WITH (MEMORY_OPTIMIZED = ON);
```

Uso híbrido:

Datos históricos (5 años) → Disco
Datos recientes (1 mes) → In-Memory



2.4.3.3: Oracle In-Memory

Similar a SQL Server:

- Oracle tradicional + opción in-memory

Dual Format:

Mismo dato existe en:
1. Formato fila (disco) → Para OLTP
2. Formato columna (RAM) → Para OLAP

Automáticamente sincronizados

2.4.3.4: Amazon MemoryDB

Qué es:

- Servicio cloud managed (Redis in-memory)

Ventajas cloud:

No necesitas:
- Comprar servidores
- Instalar software
- Configurar backups

Solo pagas por uso:
\$0.20 por GB-hora

Ejemplo:

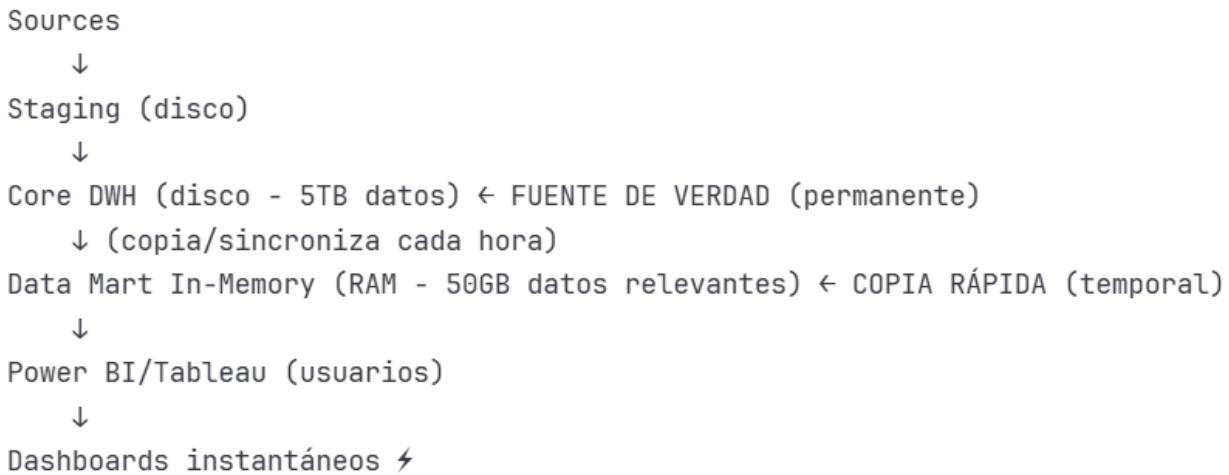
Startup necesita 50GB in-memory para cache:
Costo: $\$0.20 \times 50\text{GB} \times 24\text{h} \times 30\text{ días} = \$7,200/\text{mes}$
(vs comprar server: \$10,000 inicial + mantenimiento)

2.4.3.5: Comparacion Completa

Aspecto	Traditional DB	In-Memory DB
Velocidad query	100-500ms	1-10ms
Almacenamiento	Disco	RAM
Costo (1TB)	\$100	\$5,000-10,000
Durabilidad	<input checked="" type="checkbox"/> Nativa	<input type="checkbox"/> Requiere snapshots
Límite tamaño	Ilimitado (disco)	Limitado (RAM disponible)
Mejor para	OLTP, datos grandes	OLAP, consultas rápidas
Persistencia	<input checked="" type="checkbox"/> Automática	<input type="checkbox"/> Requiere WAL/snapshots

2.4.5: Arquitectura Típica con In-Memory

Opción 1: Data Mart In-Memory



⚠ IMPORTANTE: Data Mart In-Memory es una COPIA, NO la única fuente

Por qué funciona:

- Core tiene TODO (5TB) en disco barato (PERMANENTE)
- Data Mart tiene SOLO lo relevante (50GB) en RAM cara (TEMPORAL, REGENERABLE)
- Usuarios solo ven Data Mart (rápido)

🔑 IMPORTANTE: Persistencia del Data Mart

❓ "¿Si apago el servidor, pierdo el Data Mart?"



✓ NO - El Data Mart tiene backup en disco

Proceso:

1. Datos "viven" en RAM (velocidad)
2. Cada X minutos → Snapshot a disco (seguridad)
3. Al apagar servidor → RAM se borra
4. Al encender servidor → Carga desde disco → RAM
5. Data Mart disponible de nuevo

Resultado: NO pierdes datos

Estrategia típica de persistencia:

Opción A: Snapshots periódicos

- Cada 15-60 minutos → Copia RAM → Disco
- Al reiniciar → Carga último snapshot
- Pierdes: Solo datos entre último snapshot y apagado



Opción B: Write-Ahead Log (WAL)

- Cada cambio → Escribe log en disco primero
- Al reiniciar → Carga snapshot + aplica logs
- Pierdes: 0 datos (pero más lento)

Opción C: Recarga desde Core (MÁS COMÚN para Data Marts)

- Data Mart es "cache" del Core
- Al reiniciar → Recarga desde Core DWH
- Pierdes: 0 datos (Core tiene la verdad)

Opción 2: Híbrido

Core DWH (disco)

- └ Datos históricos 2010-2023 → Disco (lento pero OK)
- └ Datos recientes 2024 → In-Memory (rápido)



Ventaja: Balance costo/performance

2.4.6: Flujo Completo: Data Mart In-Memory

Carga Inicial (Primera vez):

Día 1, 8:00 AM - Setup inicial:



1. Core DWH (disco) tiene 5TB datos
2. Script ETL corre:
 - SELECT datos relevantes FROM Core DWH
 - WHERE fecha >= '2024-01-01' (solo datos recientes)
 - Resultado: 50GB datos
3. Cargar 50GB → Data Mart RAM
4. Tiempo: 20 minutos
5. Usuarios ahora consultan Data Mart (rápido)

Actualización Periódica (Cada hora):

Día 1, 9:00 AM - Actualización:



1. Script ETL corre automáticamente:
 - SELECT nuevos datos FROM Core DWH
 - WHERE fecha >= '2024-01-28 08:00' (solo última hora)
 - Resultado: 500MB nuevos datos
2. Actualizar Data Mart RAM con 500MB nuevos
3. Tiempo: 2 minutos
4. Usuarios continúan consultando (sin interrupciones)

Una duda sobre datamart:

⌚ Respuesta Ultra Breve:

NO, el Data Mart se carga UNA VEZ y queda disponible.

Setup (1 vez):

Core DWH → SELECT datos → Cargar Data Mart RAM

Tiempo: 20 minutos

Después (todo el dia):

Usuarios → Consultan Data Mart (ya cargado)

Tiempo: 0.5 segundos cada query

Actualización (cada X horas automáticamente):

Core DWH → SELECT solo nuevos → Actualizar Data Mart

📊 NO es así:

✗ Usuario hace query → Cargar Data Mart → Consultar → Respuesta (lento)

✓ Es así:

✓ Data Mart YA cargado (desde la mañana)
Usuario hace query → Consultar Data Mart → Respuesta (rápido)

Analogía:

- ✗ NO es abrir Word cada vez que escribes
- ✓ ES tener Word ABIERTO todo el día

Escenario: Se apaga el servidor:

Día 1, 2:00 PM - Apagón ⚡

Server se apaga

↓

RAM se borra (Data Mart desaparece)

↓

Core DWH en disco NO se afecta ✓

↓

Server reinicia

↓

Script de startup automático:

1. Detecta que Data Mart RAM está vacío
2. SELECT datos relevantes FROM Core DWH
3. Cargar 50GB → Data Mart RAM
4. Tiempo: 20 minutos

↓

Todo funciona de nuevo ✓

Datos perdidos: NINGUNO

Tiempo offline: 20 minutos

Comparación con Base de Datos Normal:

Base de Datos Normal (disco):

Apagón → Reinicio → Datos siguen en disco → 0 tiempo de recarga

Data Mart In-Memory:

Apagón → Reinicio → RAM vacía → Recarga desde Core DWH → 20 min recarga

Conclusión:

- Data Mart tiene "tiempo de recuperación" más largo
- PERO datos NO se pierden (están en Core DWH)
- Es un trade-off: Velocidad diaria vs Tiempo de recuperación

2.4.7: Ejemplo Práctico: E-commerce

Problema:

E-commerce Dashboard CEO:

- Ventas hoy en tiempo real
- Top 10 productos
- Ventas por región
- Comparación vs ayer

Datos: 10 millones transacciones/día

Solución 1: Traditional DB (lenta)

```
sql
-- Query ejecuta contra disco
SELECT
    product,
    region,
    SUM(sales) as total
FROM fact_sales
WHERE date >= '2024-01-01'
GROUP BY product, region;
```

Tiempo: 45 segundos ❌

CEO esperando... 😴

Solución 2: In-Memory DB (rápida)

Setup:

1. Core DWH (disco): 5 años histórico (5TB)
2. Data Mart In-Memory: Solo 2024 datos (50GB)
 - └ Actualizado cada hora desde Core

Dashboard CEO:

- Query contra Data Mart in-memory
- Tiempo: 0.5 segundos ✅
- CEO feliz 😊

2.4.8: Recomendaciones Prácticas + Best Practices + Resumen Ejecutivo

Cuándo usar In-Memory:

- Data Mart final (usuarios directos)
- Queries frecuentes (1000+ por minuto)
- Datos "hot" (recientes, más consultados)
- Budget disponible (\$\$\$)
- Datos caben en RAM (<500GB típicamente)

Cuándo NO usar:

- Todo el DWH (TB de datos históricos)
- OLTP (muchas escrituras)
- Budget ajustado
- Queries poco frecuentes
- Datos "cold" (históricos, rara vez consultados)

Best Practices:

1. Estrategia Híbrida

Cold data (histórico) → Disco
Hot data (reciente) → In-Memory

2. Solo lo necesario

Core: 200 tablas
Data Mart In-Memory: 20 tablas (solo relevantes)

3. Backups constantes

Snapshots cada 1 hora
Write-Ahead Log activo
Backup a disco cada 6 horas

4. Monitorear uso RAM

Si RAM >90% llena → Problema
Solución: Reducir datos o agregar RAM

Resumen Ejecutivo

In-Memory Database:

- TODO en RAM (no disco)
- 100-1000x más rápido
- 50-100x más caro
- Ideal para Data Marts y analítica en tiempo real

Trade-off:

Velocidad ⚡ vs Costo 💰

Si necesitas velocidad Y tienes presupuesto:

→ In-Memory Database

Si necesitas costo bajo:

→ Traditional Database (con optimizaciones)

Tecnologías principales:

- SAP HANA (enterprise)
- SQL Server In-Memory Tables (híbrido)
- Oracle In-Memory (híbrido)
- Amazon MemoryDB (cloud)

Arquitectura típica:

Core DWH (disco, barato, 5TB)

↓

Data Mart (in-memory, caro, 50GB)

↓

Usuarios (felices, consultas instantáneas)

2.4.9: Row-based (BDR estándar) vs Column-based (BDR para análisis/DWH)

Tradicionales: Row-based (por filas)

PostgreSQL, MySQL, Oracle, SQL Server (tradicional)

└ Almacenamiento: Fila completa junta

Modernas/Analíticas: Column-based (por columnas)

Redshift, BigQuery, Snowflake

└ Almacenamiento: Columna completa junta

BigQuery

2.4.9.1 ROW-BASED (Tradicional)

Cómo se guarda físicamente en disco:

Tabla: Empleados

id	nombre	edad	salario
1	Juan	25	3000
2	María	30	4000
3	Pedro	28	3500

En DISCO (row-based):

Bloque 1: [1, "Juan", 25, 3000]

Bloque 2: [2, "María", 30, 4000]

Bloque 3: [3, "Pedro", 28, 3500]

Cada fila COMPLETA se guarda JUNTA

2.4.9.2 COLUMN-BASED (Analítico)

Cómo se guarda físicamente en disco:

Misma tabla: Empleados

id	nombre	edad	salario
1	Juan	25	3000
2	María	30	4000
3	Pedro	28	3500

En DISCO (column-based):

Bloque 1: [1, 2, 3] ← Columna 'id'

Bloque 2: ["Juan", "María", "Pedro"] ← Columna 'nombre'

Bloque 3: [25, 30, 28] ← Columna 'edad'

Bloque 4: [3000, 4000, 3500] ← Columna 'salario'

Cada columna COMPLETA se guarda JUNTA

2.4.9.3 Ejemplo Práctico: Query

Query: "¿Cuál es el promedio de salarios?"

sql



```
SELECT AVG(salario) FROM empleados;  
```
```

#### ROW-BASED (lee TODO):

Disco lee:

Bloque 1: [1, "Juan", 25, 3000] ← Lee 4 valores, usa 1  
Bloque 2: [2, "María", 30, 4000] ← Lee 4 valores, usa 1  
Bloque 3: [3, "Pedro", 28, 3500] ← Lee 4 valores, usa 1

Total leído: 12 valores

Total usado: 3 valores (salarios)

Desperdicio: 75%

#### COLUMN-BASED (lee SOLO lo necesario):

Disco lee:

Bloque 4: [3000, 4000, 3500] ← Lee 3 valores, usa 3

Total leído: 3 valores

Total usado: 3 valores

Desperdicio: 0%

```

Imaginemos 1 millón de filas:

ROW-BASED:

```
Archivo en disco: empleados.dat

Byte 0-100: [1, "Juan", 25, 3000]
Byte 101-200: [2, "María", 30, 4000]
Byte 201-300: [3, "Pedro", 28, 3500]
...
Byte 999901-1000000: [1000000, "Ana", 35, 5000]
```

Para leer salarios:

Tienes que leer TODO el archivo (1000000 bytes)

COLUMN-BASED:

```
Archivos en disco:
```

empleados_id.dat:

Byte 0-4000000: [1, 2, 3, ..., 1000000]

empleados_nombre.dat:

Byte 0-10000000: ["Juan", "María", "Pedro", ..., "Ana"]

empleados_edad.dat:

Byte 0-4000000: [25, 30, 28, ..., 35]

empleados_salario.dat: ← SOLO ESTE

Byte 0-4000000: [3000, 4000, 3500, ..., 5000]

Para leer salarios:

Solo lees empleados_salario.dat (4000000 bytes)

Row-based = Archivo por tabla (todas las columnas mezcladas)

Column-based = Archivo por columna (cada columna separada)

2.4.10: ¿Todos los sistemas guardan sus DWH en Row-Based y cuando usamos Select para pasarlo a data mart como que se convierte a Column-Based?

Super breve. entonces todos los sistemas guardan sus DWH en Row-Based y cuando usamos Select para pasarlo a data mart como que se convierte a Column-Based.

NO exactamente.

Depende de la tecnología:

Core DWH (opciones):

- └ PostgreSQL/MySQL → Row-based
- └ Redshift/BigQuery → Column-based (ya optimizado)
- └ Snowflake → Column-based (ya optimizado)

Data Mart In-Memory:

- └ Column-based (para velocidad)

Conversión:

SI Core es row-based (PostgreSQL):

SELECT → Convierte a column-based en Data Mart ✓

SI Core es column-based (Redshift):

SELECT → Ya es column-based, solo copia ✓

Resumen:

- Core moderno = Ya es column-based
- Core viejo = Row-based, se convierte al cargar Data Mart
- NO es el SELECT quien convierte, es la **tecnología del Data Mart**

2.4.10.1: y cual es la tecnologia que convierte a column based? "SI Core es row-based (PostgreSQL): SELECT → Convierte a column-based en Data Mart ✓"

y cual es la tecnologia que convierte a column based?
"SI Core es row-based (PostgreSQL): SELECT → Convierte a column-based en Data Mart ✓ "

La tecnología del Data Mart In-Memory convierte automáticamente.

PostgreSQL (Core, row-based)

↓ SELECT datos

↓ INSERT INTO...

SAP HANA (Data Mart, column-based) ← HANA convierte al insertar

Ejemplos:

SAP HANA:

sql

```
-- SELECT saca datos de PostgreSQL (row)
-- HANA los guarda en columnar automáticamente
```

SQL Server In-Memory:

sql

```
CREATE TABLE data_mart (...)  
WITH (MEMORY_OPTIMIZED = ON,  
      COLUMNSTORE); -- Fuerza columnar  
...
```

Power BI (VertiPaq engine):

...

Import data → VertiPaq convierte a columnar automáticamente

Resumen:

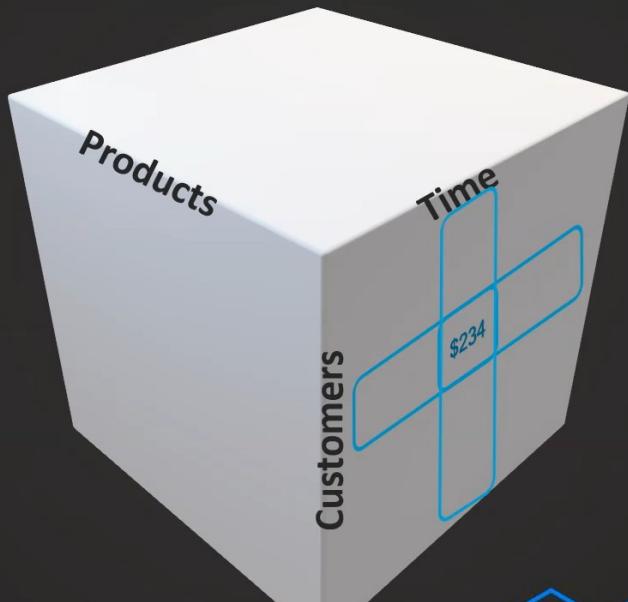
- **SELECT** solo extrae datos
- **Data Mart engine** (HANA, VertiPaq, etc) convierte a columnar al insertar

2.5: Cubes (Exhaustivo)

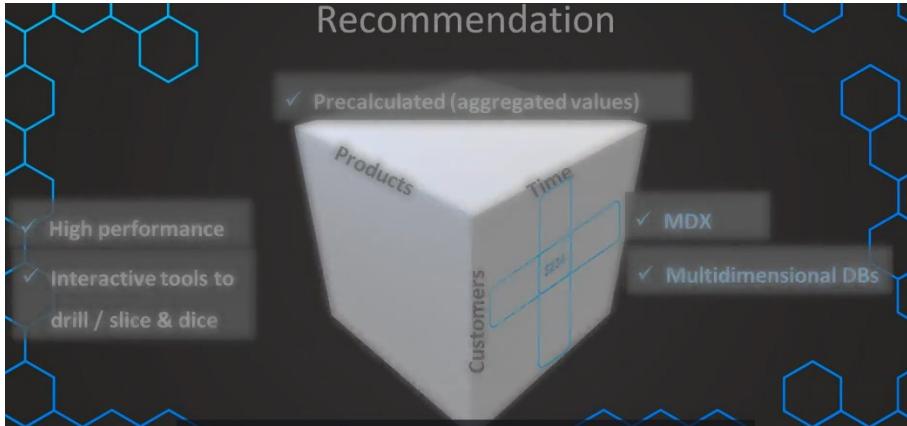
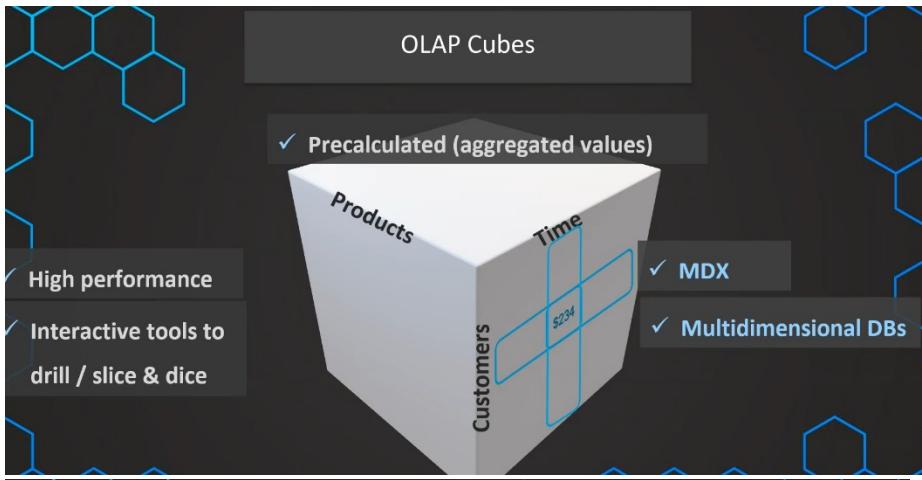
OLAP Cubes

- ✓ Traditional DWH based on relational DBMS (ROLAP)
- ✓ Data is organized non-relational in Cube (MOLAP)
Cube = Multidimensional dataset
- ✓ Arrays instead of tables
- ✓ Main reason to use: Fast query performance
- ✓ Works well with many BI solutions

OLAP Cubes



mento para obtener el importe de las ventas de ese d



- Recommendation**
- ✓ Built for a specific use-case (as data marts in general)
 - ✓ More efficient & less complex with separate data marts
 - ✓ Good for interactive queries with hierarchies
 - ✓ Optional after star schema is built in relational DB

- Alternatives**
- ✓ Less important today with advancement of hardware
 - ✓ Alternatives:
 - Tabular models (SSAS)
 - ROLAP
 - columnar storage

2.5.1 ¿Qué es un OLAP Cube?

OLAP Cube = Estructura de datos MULTIDIMENSIONAL que pre-calcula y almacena TODAS las combinaciones posibles de agregaciones.

Analogía simple:

Base de datos relacional = Hoja de cálculo (2D: filas y columnas)

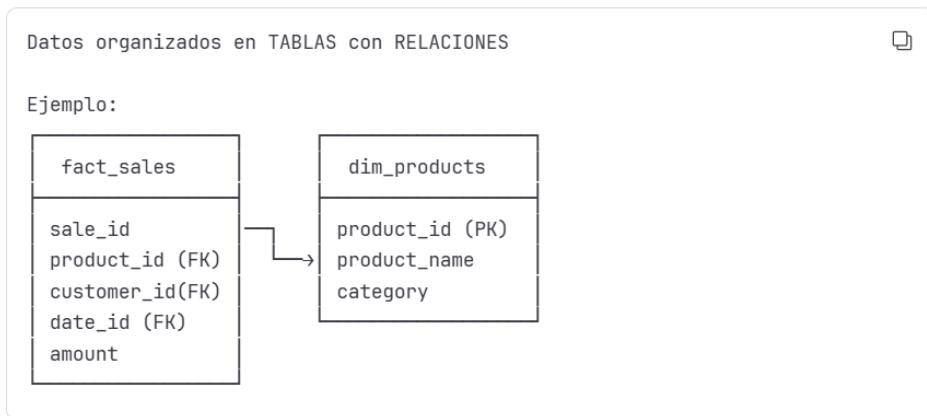
OLAP Cube = Cubo de Rubik (3D+: múltiples dimensiones)

2.5.2 ROLAP vs MOLAP – Diferencia Fundamental

2.5.2.1 ROLAP (Relational OLAP)

Qué es: Data Warehouse tradicional usando bases de datos RELACIONALES

Estructura:



Cómo funciona:



```
-- Query en tiempo real (calcula AHORA)
SELECT
    p.product_name,
    SUM(s.amount) as total_sales
FROM fact_sales s
JOIN dim_products p ON s.product_id = p.product_id
WHERE s.date_id = '2024-01-15'
GROUP BY p.product_name;
```

Tiempo: 5-30 segundos (calcula al momento)

2.5.2.2 MOLAP (Multidimensional OLAP)

Qué es: Cubo que almacena datos en estructura MULTIDIMENSIONAL (NO relacional)

Estructura:

Datos organizados en ARRAYS MULTIDIMENSIONALES

NO hay tablas, NO hay relaciones (PK/FK)

TODO pre-calculado y almacenado

Cómo funciona:

Query NO calcula NADA, solo LEE valor pre-calculado

Usuario pregunta: "Ventas de Laptop en Enero 2024"

Cubo responde: [Lee posición del array] → \$234,000

Tiempo: 0.1 segundos (solo lectura)

2.5.3 Representación Física: Cómo se VEN los Datos - row-based o colum-based

Escenario: Mismos datos de ventas en 3 formatos

Datos originales (6 transacciones):

1. Alice compró Laptop por \$1,000 en Enero en LA
2. Alice compró Mouse por \$20 en Enero en LA
3. Bob compró Laptop por \$1,000 en Enero en NY
4. Alice compró Mouse por \$20 en Febrero en LA
5. Bob compró Keyboard por \$50 en Febrero en NY
6. Alice compró Laptop por \$1,000 en Marzo en LA

2.5.3.1: 1. ROLAP - Representación Física (row-based o column-based)

Cómo se almacena: TABLAS RELACIONALES en DISCO

⚠ IMPORTANTE: ROLAP puede ser ROW-BASED o COLUMN-BASED

2.5.3.1.1: ROLAP Tradicional: ROW-BASED (Postgre, MySQL, SQL Server, Oracle)

Tecnologías: PostgreSQL, MySQL, SQL Server tradicional, Oracle tradicional

Archivo en disco: fact_sales.dat

Almacenamiento ROW-BASED (fila por fila):



```
Byte 0-100: [1, 101, 201, 301, 1000, '2024-01-15']
    └ Toda la fila JUNTA
Byte 101-200: [2, 102, 201, 301, 20, '2024-01-15']
    └ Toda la fila JUNTA
Byte 201-300: [3, 101, 202, 302, 1000, '2024-01-16']
Byte 301-400: [4, 102, 201, 301, 20, '2024-02-10']
Byte 401-500: [5, 103, 202, 302, 50, '2024-02-11']
Byte 501-600: [6, 101, 201, 301, 1000, '2024-03-05']
```

Cada FILA COMPLETA guardada JUNTA en bytes secuenciales

Desventaja para análisis:

```
Query: "SELECT SUM(amount) FROM fact_sales"
```



Tiene que leer:

Byte 0-100: [1, 101, 201, 301, 1000, '2024-01-15'] ← Lee TODO, usa
solo amount

Byte 101-200: [2, 102, 201, 301, 20, '2024-01-15'] ← Lee TODO, usa
solo amount

Byte 201-300: [3, 101, 202, 302, 1000, '2024-01-16'] ← Lee TODO, usa
solo amount

...

Desperdicio: Lee 600 bytes, usa solo 48 bytes (8 bytes × 6 valores
amount)

Eficiencia: 8%

2.5.3.1.2: ROLAP Moderno: COLUMN-BASED (Amazon Redshift, BigQuery, Snowflake, ClickHouse)

Tecnologías: Amazon Redshift, Google BigQuery, Snowflake, ClickHouse

Archivos en disco:

```
fact_sales/ (directorio con múltiples archivos)
├── column_id.dat
[1, 2, 3, 4, 5, 6]
Tamaño: 48 bytes

└── column_product_id.dat
[101, 102, 101, 102, 103, 101]
Tamaño: 48 bytes

└── column_customer_id.dat
[201, 201, 202, 201, 202, 201]
Tamaño: 48 bytes

└── column_city_id.dat
[301, 301, 302, 301, 302, 301]
Tamaño: 48 bytes

└── column_amount.dat ← SOLO ESTA para SUM
[1000, 20, 1000, 20, 50, 1000]
Tamaño: 48 bytes

└── column_date.dat
['2024-01-15', '2024-01-15', '2024-01-16', ...]
Tamaño: 120 bytes
```

Cada COLUMNA guardada en archivo SEPARADO

Ventaja para análisis:

```
Query: "SELECT SUM(amount) FROM fact_sales"

Solo lee:
column_amount.dat: [1000, 20, 1000, 20, 50, 1000]
└ Solo 48 bytes

Desperdicio: 0 bytes
Eficiencia: 100%

Velocidad: 10-100x más rápido que row-based
```

2.5.3.1.3: Comparación ROLAP Row vs Column + Estructura de Tablas de ROLAP

Aspecto	ROLAP Row-Based	ROLAP Column-Based
Tecnología	PostgreSQL, MySQL	Redshift, BigQuery, Snowflake
Almacenamiento	Fila completa junta	Columna completa junta
Mejor para	OLTP (transacciones)	OLAP (análisis)
Query analítica	Lenta (lee TODO)	Rápida (lee solo columnas necesarias)
Query transaccional	Rápida (lee fila completa)	Lenta (reconstruye fila)
Compresión	Baja (datos variados)	Alta (datos similares)
Uso DWH	⚠ Funciona pero lento	✓ Ideal

Estructura de tablas:

The screenshot shows the structure of four tables in a database named sales_dwh/:

- fact_sales.dat (tabla hechos)**:
A table with columns id, product_id, customer_id, city_id, amount, and date. It contains 6 rows of data:

id	product_id	customer_id	city_id	amount	date
1	101	201	301	1000	2024-01-15
2	102	201	301	20	2024-01-15
3	101	202	302	1000	2024-01-16
4	102	201	301	20	2024-02-10
5	103	202	302	50	2024-02-11
6	101	201	301	1000	2024-03-05

Tamaño: 600 bytes
- dim_products.dat (tabla dimensión)**:
A table with columns product_id, name, and category. It contains 3 rows of data:

product_id	name	category
101	Laptop	Electronics
102	Mouse	Electronics
103	Keyboard	Electronics

Tamaño: 200 bytes
- dim_customers.dat**:
A table with columns customer_id and name. It contains 2 rows of data:

customer_id	name
201	Alice
202	Bob

Tamaño: 100 bytes
- dim_cities.dat**:
A table with columns city_id and name. It contains 2 rows of data:

city_id	name
301	LA
302	NY

Tamaño: 80 bytes

TOTAL TAMAÑO DISCO: 980 bytes

Query ROLAP (tiempo real):



```
-- Usuario pregunta: "¿Ventas totales de Alice?"  
SELECT SUM(amount)  
FROM fact_sales f  
JOIN dim_customers c ON f.customer_id = c.customer_id  
WHERE c.name = 'Alice';
```

Proceso:

1. Leer fact_sales.dat (600 bytes)
2. Leer dim_customers.dat (100 bytes)
3. JOIN en memoria
4. Filtrar name = 'Alice'
5. SUM(amount) → calcular ahora
6. Resultado: \$2,040

Tiempo: 5-30 segundos (con millones de filas)

2.5.3.2: 2. MOLAP - Representación Física (NO ES row-based ni columna-based: ES ARRAY BASED)

Cómo se almacena: **ARRAY MULTIDIMENSIONAL** en archivo binario especializado.

⚠ IMPORTANTE: MOLAP NO es ROW-BASED ni COLUMN-BASED - **Es ARRAY-BASED (Multidimensional)**

Pero internamente puede usar COLUMN-BASED para comprimir dimensiones

2.5.3.2.1: Estructura MOLAP: Arrays N-Dimensionales

Archivo en disco: sales_cube.mdb

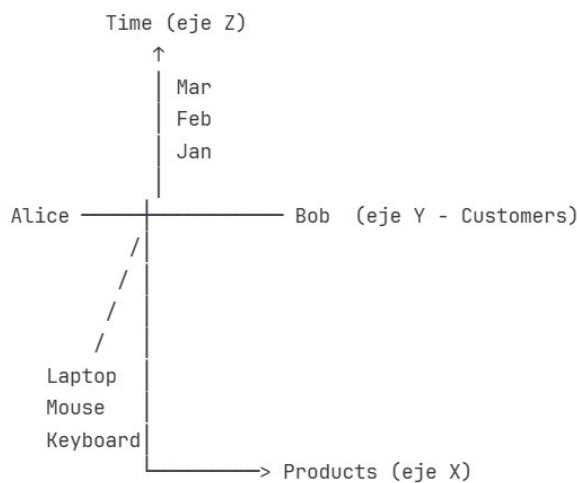
Almacenamiento ARRAY MULTIDIMENSIONAL:



NO hay concepto de "filas" o "columnas"

HAY concepto de "celdas" en espacio N-dimensional

Visualización conceptual (3D):



Cada punto en este espacio 3D = 1 celda = 1 valor pre-calculado

2.5.3.2.2: Representación en Memoria/Disco

METADATA SECTION:

Cube Name: Sales Cube
Dimensions: 3
- Products: [Laptop, Mouse, Keyboard]
- Customers: [Alice, Bob]
- Time: [Jan, Feb, Mar]
Total Cells: $3 \times 2 \times 3 = 18$ celdas base
Total with Aggregations: 54 celdas



DIMENSION INDEX (COLUMNAR internamente):

Products dimension:
Keys: [0:'Laptop', 1:'Mouse', 2:'Keyboard']
Stored column-based for compression

Customers dimension:
Keys: [0:'Alice', 1:'Bob']

Time dimension:
Keys: [0:'Jan', 1:'Feb', 2:'Mar']

```
DATA ARRAY (valores pre-calculados):
```

```
Acceso: Array[product_index][customer_index][time_index]

Array[0][0][0] = 1000 ← Laptop-Alice-Jan
Array[1][0][0] = 20   ← Mouse-Alice-Jan
Array[2][0][0] = 0    ← Keyboard-Alice-Jan
Array[0][1][0] = 1000 ← Laptop-Bob-Jan
Array[1][1][0] = 0    ← Mouse-Bob-Jan
Array[2][1][0] = 0    ← Keyboard-Bob-Jan

Array[0][0][1] = 0    ← Laptop-Alice-Feb
Array[1][0][1] = 20   ← Mouse-Alice-Feb
Array[2][0][1] = 0    ← Keyboard-Alice-Feb
Array[0][1][1] = 0    ← Laptop-Bob-Feb
Array[1][1][1] = 0    ← Mouse-Bob-Feb
Array[2][1][1] = 50   ← Keyboard-Bob-Feb
| ...
| 
| Array[0][0][2] = 1000 ← Laptop-Alice-Mar
| Array[1][0][2] = 0    ← Mouse-Alice-Mar
| Array[2][0][2] = 0    ← Keyboard-Alice-Mar
| Array[0][1][2] = 0    ← Laptop-Bob-Mar
| Array[1][1][2] = 0    ← Mouse-Bob-Mar
| Array[2][1][2] = 0    ← Keyboard-Bob-Mar
| ...
| --- AGGREGATIONS (también array-based) ---
| AggArray[0][0]['ALL'] = 2000 ← Laptop-Alice-AllTime
| AggArray[1][0]['ALL'] = 40   ← Mouse-Alice-AllTime
| ...
| AggArray['ALL'][0]['ALL'] = 2040 ← AllProducts-Alice-AllTime
```

TOTAL TAMAÑO DISCO: 2,500 bytes

2.5.3.2.3: MOLAP NO es Row ni Column - Es ARRAY (PRECALCULADO) + Accesor a Datos + Compresión Interna en MOLAP

Diferencia clave:

ROW-BASED (PostgreSQL):

```
record1 = [id=1, product=101, customer=201, amount=1000]  
record2 = [id=2, product=102, customer=201, amount=20]  
└ Cada REGISTRO es una unidad
```



COLUMN-BASED (Redshift):

```
column_id = [1, 2, 3, 4, 5, 6]  
column_product = [101, 102, 101, 102, 103, 101]  
column_amount = [1000, 20, 1000, 20, 50, 1000]  
└ Cada COLUMNNA es una unidad
```

ARRAY-BASED / MULTIDIMENSIONAL (MOLAP):

```
cube[product][customer][time] = value  
cube[0][0][0] = 1000 ← Laptop-Alice-Jan  
cube[1][0][0] = 20   ← Mouse-Alice-Jan  
└ Cada CELDA es una unidad en espacio N-dimensional  
    NO hay concepto de "fila" o "registro"
```

Acceso a Datos:

Row-based:

```
Leer fila 3: Lee bytes 201-300
```

Column-based:

```
Leer columna amount: Lee column_amount.dat completo
```

Array-based (MOLAP):

```
Leer celda [Laptop][Alice][Jan]:  
Calcula offset: (0 * 2 * 3) + (0 * 3) + 0 = 0  
Lee byte en posición 0 del array de datos
```

Compresión Interna en MOLAP:

Aunque el cubo es multidimensional, internamente puede usar técnicas COLUMNARES para comprimir:



Dimensión Products (muchas celdas = "Laptop"):

Sin comprimir: "Laptop" repetido 1000 veces = 7000 bytes

Con compresión columnar: Dictionary + pointers = 100 bytes

Valores numéricos repetidos:

Sin comprimir: [0, 0, 0, 1000, 0, 0, 0, 1000, ...]

Con compresión: Run-length encoding → mucho más pequeño

Resultado:

Cubo lógico: 2500 bytes sin comprimir

Cubo físico: 800 bytes con compresión columnar interna

[transcripción]:

Aunque el cubo es multidimensional, internamente puede usar técnicas COLUMNARES para comprimir:

Dimension Products (muchas celdas = "laptop"):

Sin Comprimir: "Laptop" repetido 1000 veces = 7000 bytes

Con comprensión columnar: Dictionary + pointers = 100 bytes

Valores numéricos repetidos:

Sin comprimir: [0, 0, 0, 1000, 0, 0, 0, 1000, ...]

Con comprensión: Run-lenght encoding -> mucho mas pequeño

Resultado:

Cubo lógico: 2500 bytes sin comprimir

Cubo físico: 800 bytes con comprensión columnar interna

2.5.3.2.4: Comparación: Row vs Column vs Array + Query MOLAP (Lectura directa)

Comparación: Row vs Column vs Array

Aspecto	Row-Based	Column-Based	Array-Based (MOLAP)
Estructura	Filas (records)	Columnas	Celdas N-dimensionales
Unidad básica	1 registro completo	1 columna completa	1 celda (coordenadas)
Acceso	Por fila	Por columna	Por coordenadas [x][y] [z]
Mejor para	OLTP	OLAP queries	OLAP drill-down
Agregaciones	✗ Calcula ahora	✗ Calcula ahora	✓ Pre-calculadas
Dimensionalidad	2D (filas×columnas)	2D (columnas×filas)	N-D (dim1×dim2×...×dimN)
Compresión	Baja	Alta	Muy alta (con sparse)

Query MOLAP (lectura directa):

```
mdx
-- Usuario pregunta: "¿Ventas totales de Alice?"
SELECT [Measures].[Sales] ON COLUMNS
FROM [Sales Cube]
WHERE [Customers].[Customer].[Alice]
```

Proceso:

1. Lookup directo: cube[ALL][Alice][ALL]
2. Leer valor: 2040
3. Resultado: \$2,040

Tiempo: 0.01-0.1 segundos (solo lectura)

2.5.3.2.3.1 Aclarando dudas: PARENTESIS: sobre el Offset y Comparacion: Para convertir coordenadas 3D -> posición en memoria Lineal (1D) para buscar la posición- PROFUNDIZAR

Array-based (MOLAP):

```
Leer celda [Laptop][Alice][Jan]:  
Calcula offset: (0 * 2 * 3) + (0 * 3) + 0 = 0  
Lee byte en posición 0 del array de datos
```

La Multiplicación: $(0 * 2 * 3) + (0 * 3) + 0 = 0$

Es para convertir coordenadas 3D -> posición en memoria lineal (1D).

Setup:

```
Array[product][customer][time]  
- Products: 3 valores (Laptop=0, Mouse=1, Keyboard=2)  
- Customers: 2 valores (Alice=0, Bob=1)  
- Time: 3 valores (Jan=0, Feb=1, Mar=2)
```

Quieres acceder: [Laptop][Alice][Jan] = [0][0][0]

Formula:

```
offset = (product * customers_total * time_total) +  
        (customer * time_total) +  
        time  
  
offset = (0 * 2 * 3) + (0 * 3) + 0 = 0
```

Por que: Memoria es LINEAL (byte 0,1,2,3...), no 3D. la formula traduce [x][y][z] -> posición en bytes.

Otro ejemplo: [Mouse][Bob][Feb] = [1][1][1]

```
offset = (1 * 2 * 3) + (1 * 3) + 1  
       = 6 + 3 + 1  
       = 10
```

Posición byte 10 en el array

2.5.3.2.3.2: Aclarando dudas: MOLAP vs ROLAP Column - NO son lo mismo + Diferencia Clave

ROLAP Column-based:

```
column_amount.dat: [1000, 20, 1000, 20, 50, 1000]  
column_product.dat: [Laptop, Mouse, Laptop, ...]
```

- Son COLUMNAS separadas (vectores 1D)
- NO hay agregaciones pre-calculadas
- Query "SUM(amount)" → CALCULA suma ahora

MOLAP Array-based:

```
Array[product][customer][time] = valor
```

- Son COORDENADAS en espacio N-D
- Agregaciones YA guardadas:
 - Array[Laptop][Alice][Jan] = 1000
 - Array[ALL][Alice][ALL] = 2040 ← PRE-CALCULADO
- Query "Total Alice" → LEE valor (no calcula)

Resultado: MOLAP 100x más rápido pero ocupa 10-100x más espacio.

2.5.3.2.3.2.1: Sobre si MOLAP es un vector/array de N dimensiones (puede ser >3D) +

Ejemplos

Aclaración Clave:

Vector matemático:

- Dimensiones = coordenadas espaciales (x, y, z)
- 3D = espacio físico (ancho, alto, profundidad)

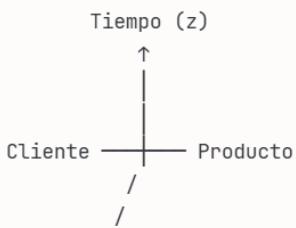
MOLAP array:

- Dimensiones = categorías de negocio
- 3D = [producto, cliente, tiempo]
- 4D = [producto, cliente, tiempo, región]
- 5D = [producto, cliente, tiempo, región, canal]
- ND = cualquier número de dimensiones

Ejemplos:

MOLAP 3D (podemos dibujar):

Array[producto][cliente][tiempo]



MOLAP 4D (NO podemos dibujar, pero existe):

Array[producto][cliente][tiempo][región]

Conceptualmente:

- 4 "ejes" independientes
- Cada combinación = 1 celda
- Imposible dibujar en papel

Ejemplo: [Laptop][Alice][Enero][California] = \$1,500

MOLAP 7D (común en enterprise):

Array[producto][cliente][tiempo][región][canal][tienda][promoción]

Ejemplo real:

[Laptop][Corp_123][Q1_2024][West][Online][Amazon][BlackFriday] = \$250K

Total dimensiones: 7

Visualización: IMPOSIBLE (solo matemáticamente)

2.5.3.2.3.2.2: Diferencia con Vector Matemático + Respuesta Directa

Concepto	Vector Matemático	MOLAP Array
Dimensión 1	x (horizontal)	Producto
Dimensión 2	y (vertical)	Cliente
Dimensión 3	z (profundidad)	Tiempo
Dimensión 4	✗ No existe en espacio físico	Región
Dimensión 5+	✗ No podemos visualizar	Canal, Tienda, etc.
Qué representa	Posición en espacio	Coordenadas en "espacio de negocio"
Visualizable	Solo hasta 3D	Solo hasta 3D

Respuesta Directa:

Sí, MOLAP es un array de N dimensiones (puede ser 3D, 4D, 5D, 10D, etc.)

MOLAP 3D: Visualizable (cubo)
MOLAP 4D+: NO visualizable (hipercubo matemático)

Límite práctico: 8-10 dimensiones
(más allá explota exponencialmente)

Analogía:

- Vector 3D = punto en habitación (x, y, z)
- MOLAP 3D = punto en "cubo de negocio" (producto, cliente, tiempo)
- MOLAP 7D = punto en "hipercubo de negocio" (7 categorías)

2.5.3.2.3.3: AD: Comprensión Columnar en MOLAP – Visualización

Compresión Interna en MOLAP:

Aunque el cubo es multidimensional, internamente puede usar técnicas COLUMNARES para comprimir:

Dimensión Products (muchas celdas = "Laptop"):
Sin comprimir: "Laptop" repetido 1000 veces = 7000 bytes
Con compresión columnar: Dictionary + pointers = 100 bytes

Valores numéricos repetidos:
Sin comprimir: [0, 0, 0, 1000, 0, 0, 0, 1000, ...]
Con compresión: Run-length encoding → mucho más pequeño

Resultado:
Cubo lógico: 2500 bytes sin comprimir
Cubo físico: 800 bytes con compresión columnar interna

Ejemplo: Dimensión Products con "Laptop" repetido 1000 veces

X SIN Compresión (tradicional):

Array multidimensional guardando cada valor:

Celda 0: "Laptop" (7 bytes)
Celda 1: "Laptop" (7 bytes)
Celda 2: "Laptop" (7 bytes)
Celda 3: "Laptop" (7 bytes)
...
Celda 999: "Laptop" (7 bytes)

Total: 1000 celdas × 7 bytes = 7,000 bytes

✓ CON Compresión Dictionary + Pointers:

Paso 1: Crear Dictionary (vocabulario único)

DICTIONARY		
ID	Valor	Bytes
0	"Laptop"	7 bytes
1	"Mouse"	5 bytes
2	"Keyboard"	8 bytes

Total dictionary: 20 bytes (solo 3 strings únicos)

Paso 2: Reemplazar valores por IDs (pointers)

Array multidimensional ahora guarda solo IDs:



```
Celda 0: 0 (apunta a "Laptop") → 1 byte  
Celda 1: 0 (apunta a "Laptop") → 1 byte  
Celda 2: 0 (apunta a "Laptop") → 1 byte  
Celda 3: 0 (apunta a "Laptop") → 1 byte  
...  
Celda 500: 1 (apunta a "Mouse") → 1 byte  
Celda 501: 0 (apunta a "Laptop") → 1 byte  
...  
Celda 999: 0 (apunta a "Laptop") → 1 byte
```

Total pointers: $1000 \text{ celdas} \times 1 \text{ byte} = 1,000 \text{ bytes}$

Paso 3: Resultado Final

MOLAP con compresión columnar



```
Dictionary: 20 bytes  
Pointers: 1,000 bytes  
TOTAL: 1,020 bytes
```

vs

MOLAP sin compresión



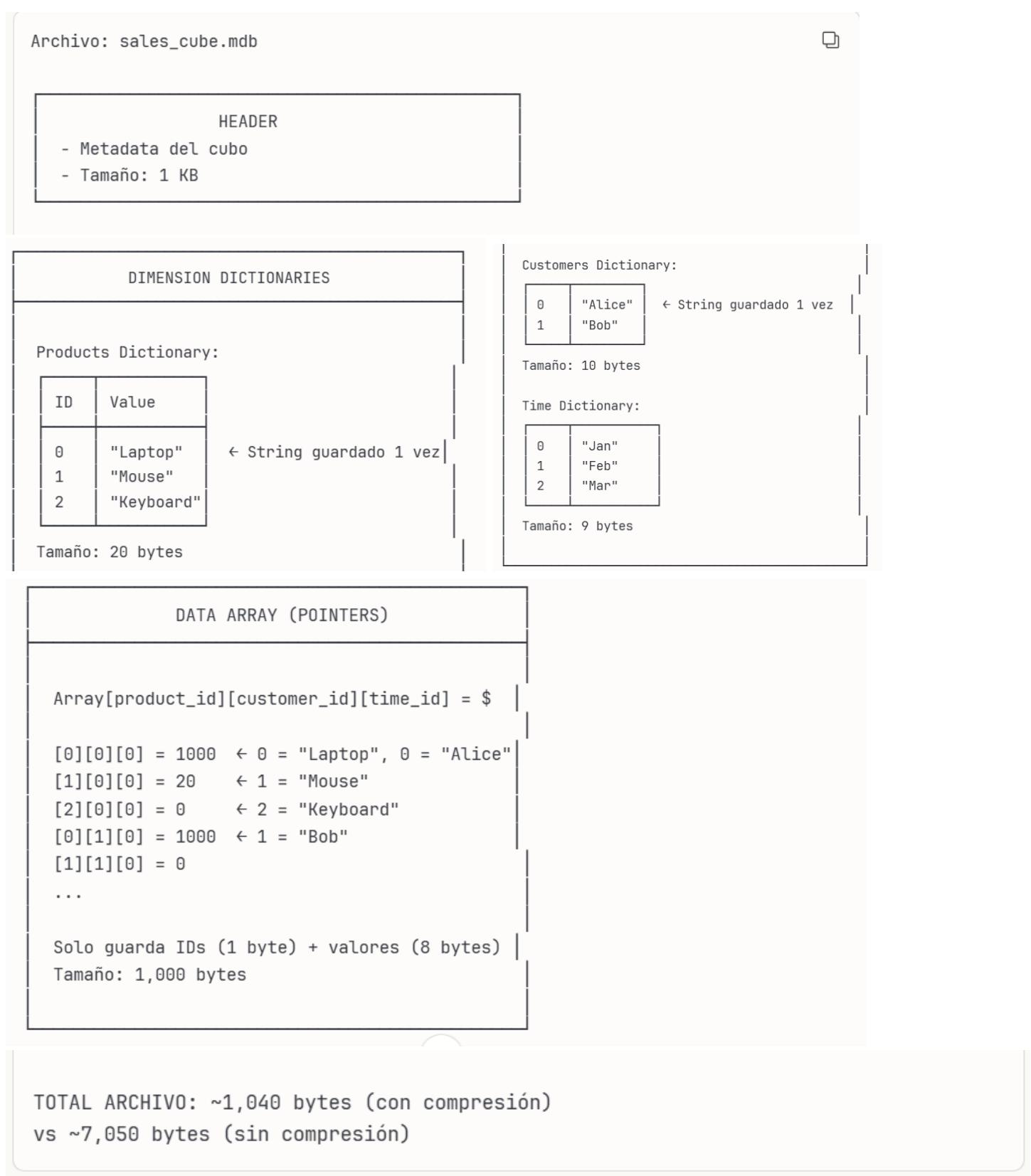
```
Strings: 7,000 bytes
```

Ahorro: $7,000 - 1,020 = 5,980 \text{ bytes}$ (85% reducción)

2.5.3.2.3.3.1: IMPORTANTE: ½ CON Compresión Dictionary + Pointers: Visualización

Completa

Estructura física en disco:



2.5.3.2.3.3.2: ½ CON Compresión Dictionary + Como Funciona en Query

Query: "Ventas de Laptop en Enero para Alice"

Paso 1: Traducir strings → IDs usando dictionaries



"Laptop" → busca en Products Dictionary → ID = 0
"Alice" → busca en Customers Dictionary → ID = 0
"Jan" → busca en Time Dictionary → ID = 0

Paso 2: Acceder array con IDs

Array[0][0][0] = 1000

Paso 3: Devolver resultado

\$1,000

Tiempo total: 0.01 segundos
(Dictionary lookup = instant en RAM)

2.5.3.2.3.3.3: 2/2 Compresión Run-Length Encoding (valores numéricos)

Ejemplo: Valores repetidos

 SIN Compresión:

```
[0, 0, 0, 1000, 0, 0, 0, 1000, 0, 0, 0, 1000]  
12 valores × 8 bytes = 96 bytes
```

 CON Run-Length Encoding:

```
[(0, count=3), (1000, count=1), (0, count=3), (1000, count=1), ...]
```



Representación comprimida:

Valor	Count
0	3
1000	1
0	3
1000	1
0	3
1000	1

6 pares × 4 bytes = 24 bytes

Ahorro: 96 - 24 = 72 bytes (75% reducción)

2.5.3.2.3.3.4: 2/2 Resultado Final (Para Compresión Dictionary + Run-Length Encoding)

Cubo LÓGICO (sin compresión):

- └─ Strings: 7,000 bytes
- └─ Valores numéricos: 96 bytes
- └─ Total: ~2,500 bytes



Cubo FÍSICO (con compresión columnar):

- └─ Dictionaries: 39 bytes
- └─ Pointers: 1,000 bytes
- └─ Valores comprimidos: 24 bytes
- └─ Total: ~800 bytes

Ratio compresión: $2,500 / 800 = 3.1:1$ (68% reducción)

Por eso MOLAP es tan eficiente: usa técnicas columnares para comprimir internamente, aunque conceptualmente sea un array multidimensional.

2.5.3.2.3.3.4.1: 1/3 ⚠ CRÍTICO: Diferencia entre Dimensión Matemática vs Dimensión

MOLAP

Contexto MATEMÁTICO (Álgebra Lineal):

```
A = [1 4 7]
    [2 5 8]
    [3 6 9]
```

Dimensión de la matriz: 3x3 (3 filas, 3 columnas)

Vector columna 1: [1] ← Vector de 3 dimensiones
[2]
[3]

Vector columna 2: [4] ← Vector de 3 dimensiones
[5]
[6]

Vector columna 3: [7] ← Vector de 3 dimensiones
[8]
[9]

Dimensión del vector = número de componentes

- Cada vector columna tiene 3 dimensiones (3 componentes)
- Valores: SOLO números (int, float, complex)

✓ Sí, grupos de vectores columna forman ESPACIOS VECTORIALES

Contexto MOLAP (Data Warehouse):

"Dimensión" ≠ "Dimensión matemática"



En MOLAP:

Dimensión = Categoría de negocio

Array[producto][cliente][tiempo]
↓ ↓ ↓
Dimensión Dimensión Dimensión
(Products) (Customers) (Time)

3 dimensiones MOLAP ≠ Vector 3D matemático

Comparación Directa

Matriz Matemática:

```
A = [1 4 7]
    [2 5 8]
    [3 6 9]
```

Fila 1: [1, 4, 7] ← NO es un vector de "3 dimensiones"
Es un VECTOR FILA con 3 componentes

Columna 1: [1] ← Si es un vector de 3 dimensiones
[2] (3 componentes verticales)
[3]

Lectura:

- A[fila][columna]
- A[0][0] = 1 (fila 0, columna 0)
- A[2][1] = 6 (fila 2, columna 1)

Valores: SOLO números

Array MOLAP:

```
Array[producto][cliente][tiempo] = valor
```



```
Array[0][0][0] = 1000
Array[1][0][0] = 20
Array[2][0][0] = 0
```

NO es una matriz matemática
Es un array multidimensional de negocio

Cada "dimensión" = categoría:

- producto: [Laptop=0, Mouse=1, Keyboard=2]
- cliente: [Alice=0, Bob=1]
- tiempo: [Jan=0, Feb=1, Mar=0]

Valores: Puede ser int, float, string, cualquier cosa

Aclaraciones Clave

1. Dimensión en Vector Matemático:

```
Vector columna v:  
v = [1] ← 3 dimensiones (3 componentes)  
[2]  
[3]
```

Dimensión = número de filas (componentes)
NO es la fila [1, 2, 3]

2. Fila vs Columna:

```
Matriz A:  
[1 4 7] ← Fila 1 (vector fila de 3 elementos)  
[2 5 8] ← Fila 2  
[3 6 9] ← Fila 3  
↑ ↑ ↑  
Col Col Col  
1 2 3
```

```
Vector FILA: [1, 4, 7] (horizontal)  
Vector COLUMNAS: [1] (vertical)  
[2]  
[3]
```

En álgebra lineal:
"Vector de N dimensiones" típicamente = vector COLUMNAS con N componentes

3. Espacios Vectoriales:

Sí, conjunto de vectores columna puede formar espacio vectorial

Ejemplo:

```
v1 = [1]    v2 = [4]    v3 = [7]  
[2]        [5]        [8]  
[3]        [6]        [9]
```

Si son linealmente independientes:

→ Forman base de espacio vectorial R^3
→ Cualquier vector en R^3 se puede escribir como combinación de v1, v2, v3

4. MOLAP NO usa Vectores Matemáticos:

INCORRECTO pensar:
"Fila [1, 2, 3] = 3 dimensiones de producto, cliente, tiempo"

CORRECTO:
MOLAP Array[producto][cliente][tiempo]

[1, 2, 3] NO es un vector
Es una COORDENADA (tupla de índices):
- producto = 1 (Mouse)
- cliente = 2 (tercer cliente)
- tiempo = 3 (cuarto mes)

Array[1][2][3] = valor en esa coordenada

2.5.3.2.3.3.4.2: 2/3 Ejemplo Completo

Matriz Matemática (Álgebra Lineal):

```
A = [100 200 300]
    [400 500 600]

Dimensión matriz: 2x3 (2 filas, 3 columnas)

Vector columna 1:
v1 = [100] ← Vector de 2 dimensiones
      [400] (2 componentes)

Vector columna 2:
v2 = [200] ← Vector de 2 dimensiones
      [500]

Vector columna 3:
v3 = [300] ← Vector de 2 dimensiones
      [600]

Acceso: A[fila][columna]
A[0][1] = 200 (fila 0, columna 1)

Valores: Solo números (int, float)
```

Array MOLAP (Data Warehouse):

```
Array[producto][región][mes]

Array[0][0][0] = "Laptop vendido en Norte en Enero"
Array[1][2][3] = "Mouse vendido en Sur en Abril"

"3 dimensiones MOLAP" = 3 categorías de negocio
NO son coordenadas espaciales (x, y, z)

Valores: Puede ser:
- Números: 1000, 20.5
- Strings: "Laptop", "California" ← ✓ Sí puede guardar strings
- Dates: "2024-01-15"
- Cualquier tipo de dato

Acceso: Array[categoría1][categoría2][categoría3]
```

⌚ Respuesta a tus Preguntas

1. "Vector columna en contexto matemático?"

✓ Sí, en álgebra lineal hablamos de vectores COLUMNA:

```
v = [1] ← Vector columna de 3 dimensiones
```

```
[2]
```

```
[3]
```

Dimensión = número de componentes (filas)

2. "¿Grupos de vectores columnares = espacios vectoriales?"

Sí, conjunto de vectores puede formar un espacio vectorial:

Si v1, v2, v3 son linealmente independientes

→ Generan (span) un espacio vectorial

→ Forman una base del espacio

3. "La fila [1,2,3] representa 3 dimensiones?"

NO en matemáticas, DEPENDE en MOLAP:

En Algebra Lineal:

```
[1, 2, 3] = Vector FILA (1 fila, 3 columnas)
```



NO es "3 dimensiones"

Es un vector con 3 componentes horizontal

Vector de 3 dimensiones sería:

```
[1] ← Columna
```

```
[2]
```

```
[3]
```

En MOLAP:

[1, 2, 3] puede ser COORDENADAS (índices):

- producto=1, cliente=2, tiempo=3

→ Acceso: Array[1][2][3]

NO es un "vector de 3 dimensiones"

Es una tupla de 3 índices para navegar el cubo

4. "¿Puede guardar strings además de ints?"

En Álgebra Lineal: NO, vectores/matrices solo números

En MOLAP: Sí, puede guardar CUALQUIER tipo:

- Integers: 1000, 50
- Floats: 1250.75
- Strings: "Laptop", "Alice"
- Dates: "2024-01-15"
- Booleans: true, false

2.5.3.2.3.3.4.2: 3/3 Tabla Resumen + Conclusión

Concepto	Álgebra Lineal	MOLAP
"Dimensión"	Número de componentes	Categoría de negocio
Vector columna	[1] [2] [3]	No aplica
Espacio vectorial	Sí (conjunto de vectores)	No aplica
Fila [1,2,3]	Vector fila (3 componentes)	Coordenadas/índices
Valores	Solo números	Cualquier tipo (int, string, etc)
3D	3 componentes (x,y,z)	3 categorías (prod, cliente, tiempo)

⌚ Conclusión

NO confundir:

- dimensión matemática (componentes de un vector)
- dimensión MOLAP (categorías de negocio)

Son conceptos DIFERENTES con el mismo nombre

2.5.3.2.3.3.5: 1/6: 📈 Cómo se Guarda el VALOR Real en MOLAP

CRÍTICO: Hay DOS tipos de datos diferentes

1. ÍNDICES (dimensiones) → usan Dictionary + IDs
2. VALORES (medidas) → guardados directamente como números

Transcripción:

1. INDICES (dimensiones) -> usan Dictionary + IDs
2. Valores (medidas) -> guardados directamente como números

Estructura Real del Array

Array[product_id][customer_id][time_id] =VALOR

DIMENSIONES (índices)	MEDIDA (valor)
[0][0][0] ↓ ↓ ↓ └── time_id=0 └── customer_id=0 └── product_id=0	→ 1000

Los ÍNDICES [0][0][0] usan Dictionary
El VALOR 1000 se guarda DIRECTO (sin Dictionary)

Los índices [0][0][0] usan Dictionary
El VALOR 1000 se guarda DIRECTO (sin dictionary)

2.5.3.2.3.3.6: 2/6: Visualización Paso a Paso

Paso 1: Guardar el dato

Dato original:

Alice compró Laptop por \$1,000 en Enero

Paso 2: Separar en Dimensiones vs Valor

DIMENSIONES (strings):

- Producto: "Laptop"
- Cliente: "Alice"
- Tiempo: "Enero"

VALOR (número):

\$1,000



Paso 3: Convertir Dimensiones a IDs (Dictionary)

Products Dictionary:

ID	Value
0	"Laptop"
1	"Mouse"
2	"Keyboard"



← "Laptop" → ID=0

Customers Dictionary:

ID	Value
0	"Alice"
1	"Bob"



← "Alice" → ID=0

Time Dictionary:

ID	Value
0	"Jan"
1	"Feb"
2	"Mar"



← "Enero" → ID=0

Resultado:

"Laptop" → 0

"Alice" → 0

"Enero" → 0

Paso 4: Guardar en Array

Array[0][0][0] = 1000

Físicamente en memoria:

CELDA EN POSICIÓN [0][0][0]

Offset calculado: 0
 $(0*2*3) + (0*3) + 0 = 0$

En esa posición guarda:

1000

← 8 bytes (int64)

El valor 1000 se guarda DIRECTO como número entero
NO usa Dictionary

Archivo Físico Completo

Archivo: sales_cube.mdb

DIMENSION DICTIONARIES
(para ÍNDICES solamente)

Products: {0:"Laptop", 1:"Mouse", 2:"Keyboard"}
Customers: {0:"Alice", 1:"Bob"}
Time: {0:"Jan", 1:"Feb", 2:"Mar"}

↓

DATA ARRAY
(valores DIRECTOS, sin Dictionary)

Posición 0: 1000 ← [0][0][0] = Alice-Laptop-Jan
Posición 1: 20 ← [1][0][0] = Alice-Mouse-Jan
Posición 2: 0 ← [2][0][0] = Alice-Keyboard-Jan
Posición 3: 1000 ← [0][1][0] = Bob-Laptop-Jan
Posición 4: 0 ← [1][1][0] = Bob-Mouse-Jan
Posición 5: 0 ← [2][1][0] = Bob-Keyboard-Jan
...

Cada valor: 8 bytes (int64)

GUARDADOS DIRECTAMENTE

Sin compresión, sin dictionary

2.5.3.2.3.3.7: 3/6 Como funciona una Query Completa

Query: "¿Cuánto vendió Alice en Laptop en Enero?"

PASO 1: Traducir strings → IDs (usando Dictionaries)



Usuario pregunta: "Alice", "Laptop", "Enero"

Buscar en Dictionaries:

- "Laptop" en Products Dictionary → ID = 0
- "Alice" en Customers Dictionary → ID = 0
- "Enero" en Time Dictionary → ID = 0

Resultado: [0][0][0]

PASO 2: Calcular posición en memoria

```
offset = (product_id * customers_total * time_total) +  
        (customer_id * time_total) +  
        time_id
```

```
offset = (0 * 2 * 3) + (0 * 3) + 0 = 0
```

Posición en array: 0

PASO 3: Leer valor DIRECTO de esa posición

Array en memoria:

Pos 0	1	2	3	4	5	...
1000	20	0	1000	0	0	...

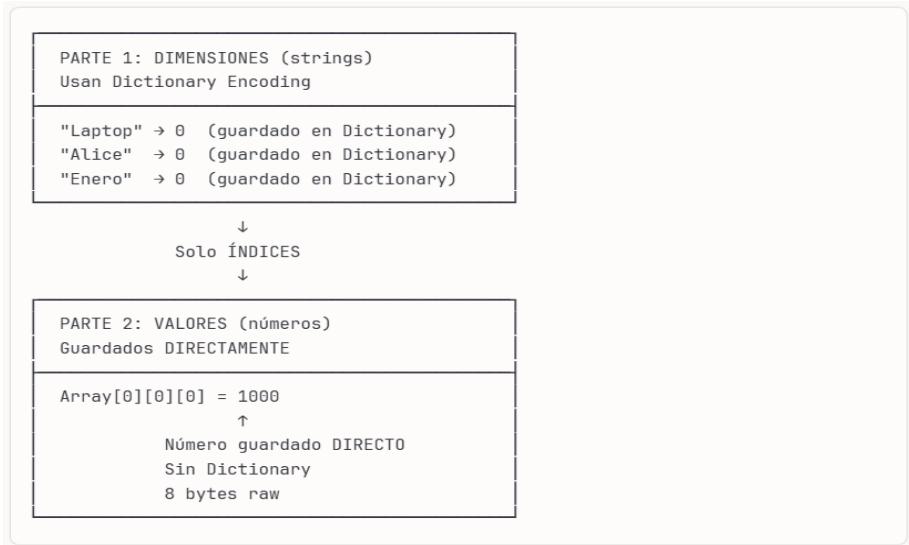
↑
Lee aquí

Valor: 1000

PASO 4: Devolver resultado

Respuesta: \$1,000

2.5.3.2.3.3.8: 4/6 Como Converge String con Int: SPOILER: No hay convergencia, Son separados



2.5.3.2.3.3.9: 6/6 Ejemplo Físico en Bytes + Resumen Final

Guardar: Alice compró Laptop por \$1,000 en Enero

```
ARCHIVO: sales_cube.mdb
```

Bytes 0-100: DICTIONARIES

Products Dictionary:
Byte 0-10: "Laptop" (string guardado)
Byte 11-21: "Mouse"
Byte 22-32: "Keyboard"

Customers Dictionary:
Byte 33-40: "Alice"
Byte 41-48: "Bob"

Time Dictionary:
Byte 49-56: "Jan"
Byte 57-64: "Feb"
Byte 65-72: "Mar"

Bytes 100-end: DATA ARRAY

Posición [0][0][0]:
Byte 100-107: [00 00 00 00 00 00 03 E8]
 ↓
 1000 en hexadecimal
 Guardado como int64 (8 bytes)

Posición [1][0][0]:
Byte 108-115: [00 00 00 00 00 00 00 14]
 ↓
 20 en hexadecimal

Posición [2][0][0]:
Byte 116-123: [00 00 00 00 00 00 00 00]
 ↓
 0 en hexadecimal

Resumen Final

Los ÍNDICES y los VALORES son TOTALMENTE SEPARADOS:

Componente	Tipo Original	Cómo se Guarda	Ubicación
product_id	String "Laptop"	Dictionary: 0 → "Laptop"	Dictionaries section
customer_id	String "Alice"	Dictionary: 0 → "Alice"	Dictionaries section
time_id	String "Enero"	Dictionary: 0 → "Enero"	Dictionaries section
VALOR	Número 1000	Directo: 1000 (8 bytes)	Data Array posición 0

NO hay "convergencia":

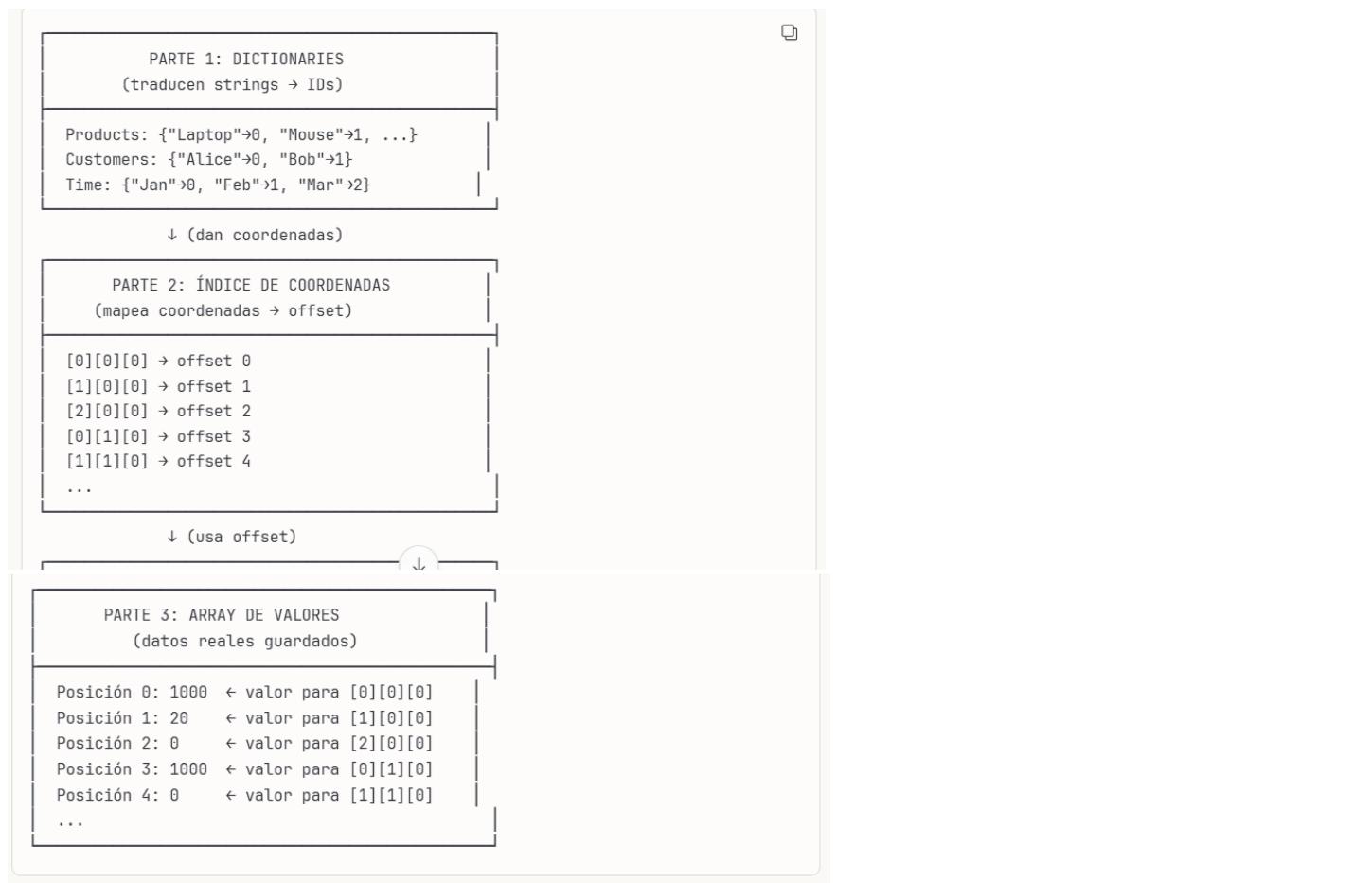
- Strings (dimensiones) → Convertidos a IDs vía Dictionary
- Numeros (valores) → Guardados directamente como bytes

Query usa:

1. Dictionaries para traducir strings → IDs
2. IDs para calcular offset
3. Offset para leer valor directo del array

2.5.3.2.3.3.10: 1/3 MOLAP guarda los valores en un ARRAY SEPARADO indexado por coordenadas + Visualización Completa

Estructura Real en Memoria



Visualización Completa

Archivo físico: sales_cube.mdb

Byte 0-99: HEADER
- Metadata

Byte 100-199: DIMENSION DICTIONARIES

Products:

ID 0 → "Laptop"
ID 1 → "Mouse"
ID 2 → "Keyboard"

Customers:

ID 0 → "Alice"
ID 1 → "Bob"

Time:

ID 0 → "Jan"
ID 1 → "Feb"
ID 2 → "Mar"

Byte 200-299: COORDINATE INDEX (opcional)

Mapeo coordenadas → offset en array de valores

[0][0][0] → 0
[1][0][0] → 1
[2][0][0] → 2
[0][1][0] → 3
...

(Este mapeo se calcula con fórmula, pero
puede estar pre-calculado para velocidad)

Byte 300-end: VALUES ARRAY
(ARRAY SEPARADO)

Offset 0: [1000] (8 bytes)	← [0][0][0]
Offset 1: [20] (8 bytes)	← [1][0][0]
Offset 2: [0] (8 bytes)	← [2][0][0]
Offset 3: [1000] (8 bytes)	← [0][1][0]
Offset 4: [0] (8 bytes)	← [1][1][0]
Offset 5: [0] (8 bytes)	← [2][1][0]

Array de valores SEPARADO e INDEXADO
Solo contiene números (int64)
1 valor = 8 bytes consecutivos

2.5.3.2.3.3.11: 2/3 Proceso Completo: Query de Principio a Fin (SUPER IMPORTANTE)

Query: "Ventas de Laptop para Alice en Enero"

PASO 1: Traducir strings → IDs (DIMENSIONES)

Entrada: "Laptop", "Alice", "Enero"

Buscar en Dictionaries:

Products Dictionary
"Laptop" → ID = 0

Customers Dictionary
"Alice" → ID = 0

Time Dictionary
"Enero" → ID = 0

Resultado: Coordenadas [0][0][0]

PASO 2: Convertir coordenadas → offset (ÍNDICE)

Fórmula:

```
offset = (product_id × customers_total × time_total) +  
       (customer_id × time_total) +  
       time_id
```

```
offset = (0 × 2 × 3) + (0 × 3) + 0  
offset = 0 + 0 + 0  
offset = 0
```

Coordenadas [0][0][0]
↓
Offset = 0

PASO 3: Usar offset para leer ARRAY DE VALORES

Values Array en memoria:

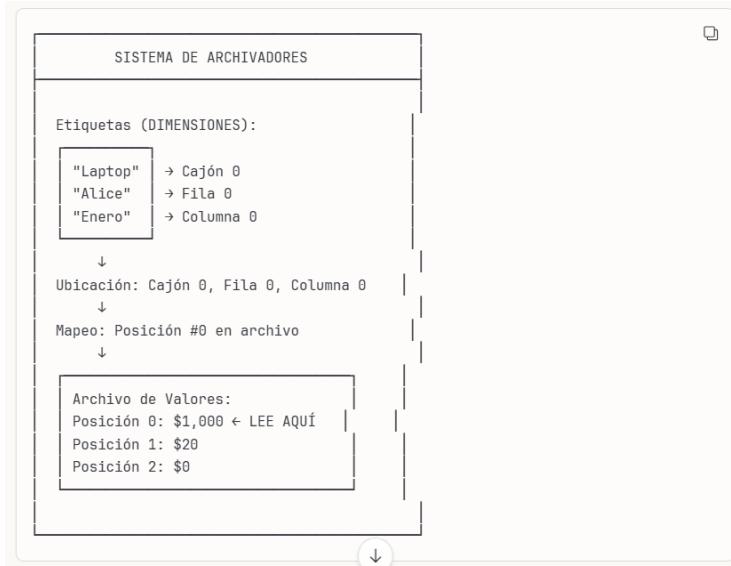
Posición	Valor	Coordenadas
0	1000	[0][0][0] ← AQUÍ
1	20	[1][0][0]
2	0	[2][0][0]
3	1000	[0][1][0]
4	0	[1][1][0]
5	0	[2][1][0]

Lee: Values[offset=0] = 1000

PASO 4: Devolver resultado

Resultado: \$1,000

2.5.3.2.3.3.12: 3/3 Analogía Perfecta – MOLAP funciona como un sistema de coordenadas + tabla de valores + Estructura en Memoria (C-style) + Respuesta final



Estructura en Memoria (C-style)

```
// Estructura conceptual en código

struct MOLAP_Cube {
    // PARTE 1: Dictionaries (dimensiones)
    Dictionary products;    // {"Laptop"→0, "Mouse"→1, ...}
    Dictionary customers;   // {"Alice"→0, "Bob"→1}
    Dictionary time;        // {"Jan"→0, "Feb"→1, "Mar"→2}

    // PARTE 2: Dimensiones del cubo
    int products_count = 3;
    int customers_count = 2;
    int time_count = 3;

    // PARTE 3: Array de valores (SEPARADO)
    int64_t* values;    // Array de 18 valores (3x2x3)
    //           ^
    //           Array INDEXADO por offset
};

// Acceso
int64_t get_value(int product_id, int customer_id, int time_id) {
    // Calcular offset
    int offset = (product_id * customers_count * time_count) +
                 (customer_id * time_count) +
                 time_id;

    // Leer del array de valores usando el offset
    return values[offset];
}

// Query: get_value(0, 0, 0)
// offset = (0*2*3) + (0*3) + 0 = 0
// return values[0] = 1000
```
`
```

## 🎯 Respuesta Directa

1. DIMENSIONES (strings)
  - ↳ DICTIONARIES
    - ↳ IDs (0, 1, 2, ...)
2. IDs (coordenadas [x][y][z])
  - ↳ FÓRMULA OFFSET
    - ↳ Posición numérica (0, 1, 2, ...)
3. POSICIÓN (offset)
  - ↳ ARRAY DE VALORES (separado)
    - ↳ VALOR REAL (1000, 20, 0, ...)

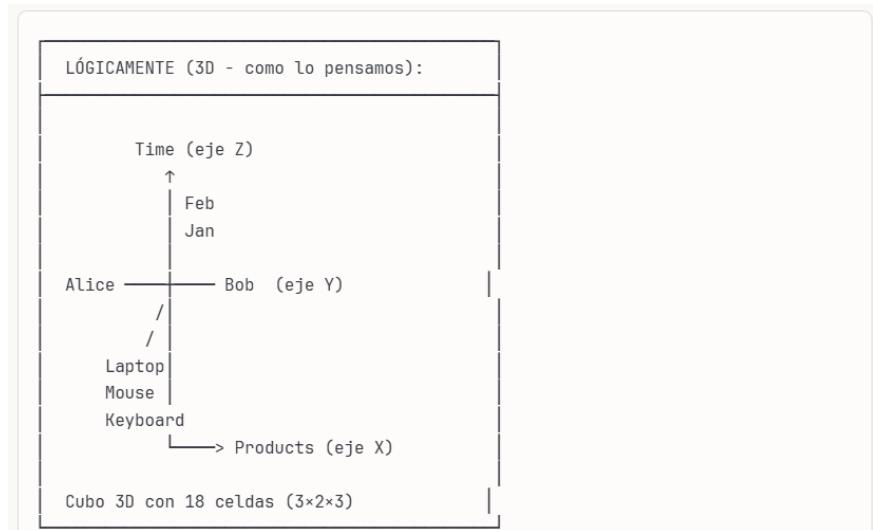
Los valores NO están "mezclados" con las dimensiones.  
Son un ARRAY SEPARADO indexado por las coordenadas.

### **2.5.3.2.3.3.13: 1/2 Sobre si el Array de Valores es Siempre 1D (Lineal) + Estructura Real**

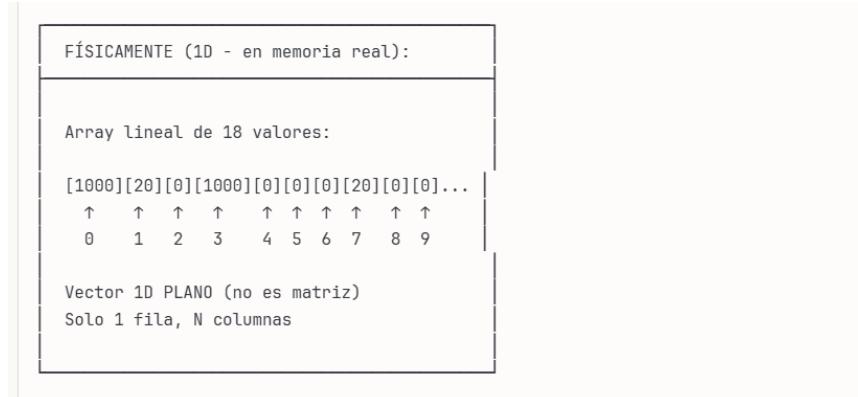
**Memoria + Mapeo 3D -> 1D**

**Critico:** NO es matriz ni vectores columna – es un ARRAY 1D PLANO

## Cómo se Ve Realmente

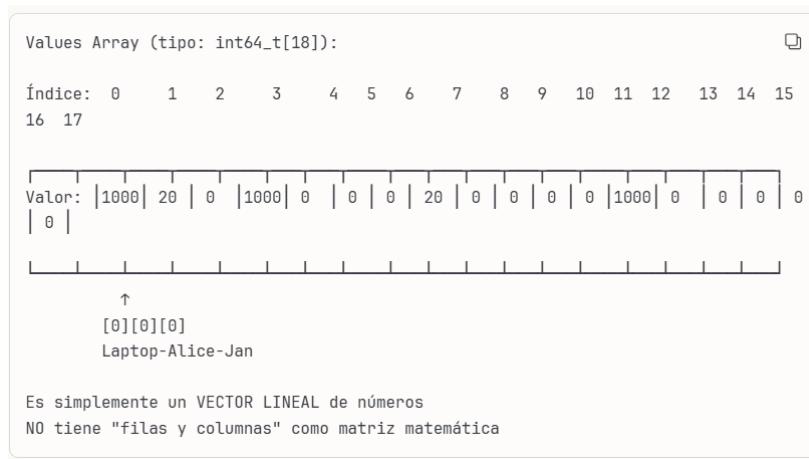


[nota: esto es una dimensión, se confundió cloude]



 Estructura Real en Memoria

## **Array de Valores = Vector 1D (NO matriz)**



## Mapeo 3D → 1D

Como las coordenadas 3D se mapean a posiciones 1D:

| Coordenadas 3D                 | Offset 1D | Valor |
|--------------------------------|-----------|-------|
| [0][0][0] (Laptop-Alice-Jan)   | → 0       | 1000  |
| [1][0][0] (Mouse-Alice-Jan)    | → 1       | 20    |
| [2][0][0] (Keyboard-Alice-Jan) | → 2       | 0     |
| [0][1][0] (Laptop-Bob-Jan)     | → 3       | 1000  |
| [1][1][0] (Mouse-Bob-Jan)      | → 4       | 0     |
| [2][1][0] (Keyboard-Bob-Jan)   | → 5       | 0     |
| [0][0][1] (Laptop-Alice-Feb)   | → 6       | 0     |
| [1][0][1] (Mouse-Alice-Feb)    | → 7       | 20    |
| [2][0][1] (Keyboard-Alice-Feb) | → 8       | 0     |
| [0][1][1] (Laptop-Bob-Feb)     | → 9       | 0     |
| [1][1][1] (Mouse-Bob-Feb)      | → 10      | 0     |
| [2][1][1] (Keyboard-Bob-Feb)   | → 11      | 50    |
| [0][0][2] (Laptop-Alice-Mar)   | → 12      | 1000  |
| [1][0][2] (Mouse-Alice-Mar)    | → 13      | 0     |
| [2][0][2] (Keyboard-Alice-Mar) | → 14      | 0     |
| [0][1][2] (Laptop-Bob-Mar)     | → 15      | 0     |
| [1][1][2] (Mouse-Bob-Mar)      | → 16      | 0     |
| [2][1][2] (Keyboard-Bob-Mar)   | → 17      | 0     |

El cubo 3D se "aplana" a un array 1D

## Visualización: 3D "Desplegado" a 1D

| CUBO 3D:           |                                                                                                                                                                                                                                                                                                           |      |    |   |      |   |    |      |   |    |      |   |   |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|----|---|------|---|----|------|---|----|------|---|---|
| Jan:               | <table border="1"><tr><td>1000</td><td>20</td><td>0</td></tr><tr><td>1000</td><td>0</td><td>0</td></tr></table>   Laptop, Mouse, Keyboard para Alice<br><table border="1"><tr><td>1000</td><td>0</td><td>0</td></tr><tr><td>1000</td><td>0</td><td>0</td></tr></table>   Laptop, Mouse, Keyboard para Bob | 1000 | 20 | 0 | 1000 | 0 | 0  | 1000 | 0 | 0  | 1000 | 0 | 0 |
| 1000               | 20                                                                                                                                                                                                                                                                                                        | 0    |    |   |      |   |    |      |   |    |      |   |   |
| 1000               | 0                                                                                                                                                                                                                                                                                                         | 0    |    |   |      |   |    |      |   |    |      |   |   |
| 1000               | 0                                                                                                                                                                                                                                                                                                         | 0    |    |   |      |   |    |      |   |    |      |   |   |
| 1000               | 0                                                                                                                                                                                                                                                                                                         | 0    |    |   |      |   |    |      |   |    |      |   |   |
| Feb:               | <table border="1"><tr><td>0</td><td>20</td><td>0</td></tr><tr><td>0</td><td>0</td><td>50</td></tr></table> Alice<br><table border="1"><tr><td>0</td><td>0</td><td>50</td></tr><tr><td>0</td><td>0</td><td>0</td></tr></table> Bob                                                                         | 0    | 20 | 0 | 0    | 0 | 50 | 0    | 0 | 50 | 0    | 0 | 0 |
| 0                  | 20                                                                                                                                                                                                                                                                                                        | 0    |    |   |      |   |    |      |   |    |      |   |   |
| 0                  | 0                                                                                                                                                                                                                                                                                                         | 50   |    |   |      |   |    |      |   |    |      |   |   |
| 0                  | 0                                                                                                                                                                                                                                                                                                         | 50   |    |   |      |   |    |      |   |    |      |   |   |
| 0                  | 0                                                                                                                                                                                                                                                                                                         | 0    |    |   |      |   |    |      |   |    |      |   |   |
| Mar:               | <table border="1"><tr><td>1000</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr></table> Alice<br><table border="1"><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr></table> Bob                                                                         | 1000 | 0  | 0 | 0    | 0 | 0  | 0    | 0 | 0  | 0    | 0 | 0 |
| 1000               | 0                                                                                                                                                                                                                                                                                                         | 0    |    |   |      |   |    |      |   |    |      |   |   |
| 0                  | 0                                                                                                                                                                                                                                                                                                         | 0    |    |   |      |   |    |      |   |    |      |   |   |
| 0                  | 0                                                                                                                                                                                                                                                                                                         | 0    |    |   |      |   |    |      |   |    |      |   |   |
| 0                  | 0                                                                                                                                                                                                                                                                                                         | 0    |    |   |      |   |    |      |   |    |      |   |   |
| ↓ "Aplanar" a 1D ↓ |                                                                                                                                                                                                                                                                                                           |      |    |   |      |   |    |      |   |    |      |   |   |

ARRAY 1D:

```
[1000,20,0,1000,0,0, 0,20,0,0,0,50, 1000,0,0,0,0]
```

└ Jan └ Feb └ Mar └

Es un VECTOR SIMPLE 1D, no matriz

## 2.5.3.2.3.3.14: 2/2 MOLAP no tiene “FACT TABLES” en el sentido relacional (tablas filas/columnas) + Diferencia Clave

MOLAP si tiene “FACTS” (valores/medidas) pero guardados en array multidimensional

### ⚠ CRÍTICO: Facts vs Fact Tables - Diferencia Clave ROLAP vs MOLAP

Aclaración Importante: Fact ≠ Fact Tables

FACTS (valores/medidas):

- Los datos numéricos reales (ventas, cantidades, montos)
- Ejemplos: \$1000, 20 unidades, 50 items
- Existen en AMBOS (ROLAP y MOLAP)

FACT TABLES (tablas de hechos relacionales):

- Estructura de tabla con filas y columnas
- Solo existe en ROLAP
- NO existe en MOLAP

### Comparación: ROLAP vs MOLAP

ROLAP (relacional)

Tiene FACT TABLE (tabla relacional):

fact\_sales (tabla con filas y columnas):

| id | product | customer | amount |
|----|---------|----------|--------|
| 1  | 101     | 201      | 1000   |
| 2  | 102     | 201      | 20     |

Facts guardados EN tabla relacional  
(estructura 2D: filas × columnas)

MOLAP (multidimensional)

NO tiene FACT TABLE (sin tablas)  
 SÍ tiene FACTS (valores)

Estructura:

1. Dictionaries (dimensiones)
2. Array 1D de FACTS

Values Array (vector 1D con facts):

[1000, 20, 0, 1000, 0, 0, ...]

↑ ↑

fact fact

Facts guardados EN array multidimensional  
(estructura ND → 1D lineal en memoria)

## 1. Donde se guardan los FACTS:

```
ROLAP:
Facts en → Fact Table (tabla relacional)
└─ Fila 1: [id=1, prod=101, cust=201, amount=1000]
└─ Fila 2: [id=2, prod=102, cust=201, amount=20]
└─ ...
```

```
MOLAP:
Facts en → Values Array (array 1D)
[1000, 20, 0, 1000, 0, 0, ...]
└─ Valores indexados por coordenadas
```

## 2. Cómo se accede a los FACTS:

```
ROLAP:
Query SQL → Scan tabla → Filtrar filas → Retornar facts
```

```
SELECT amount
FROM fact_sales
WHERE product_id = 101 AND customer_id = 201;
```

Lee filas completas hasta encontrar coincidencias

```
MOLAP:
Coordenadas → Offset → Leer fact del array
```

```
product="Laptop" → 0
customer="Alice" → 0
time="Jan" → 0

Offset = (0*2*3) + (0*3) + 0 = 0
Value = Array[0] = 1000

Acceso directo por indice
```

## 3. Estructura de almacenamiento:

### ROLAP (Fact Table):

Tabla relacional en disco:

Row 0: [1, 101, 201, 1000, '2024']  
Row 1: [2, 102, 201, 20, '2024']  
Row 2: [3, 101, 202, 1000, '2024']

Facts mezclados con dimensiones

Estructura: Filas × Columnas

### MOLAP (Array Multidimensional):

Dimensiones (separadas):

Products: {0:'Laptop', 1:'Mouse'}  
Customers: {0:'Alice', 1:'Bob'}  
Time: {0:'Jan', 1:'Feb'}

↓

↙

Facts (array separado):

[1000, 20, 0, 1000, 0, 0, ...]

Facts SEPARADOS de dimensiones

Estructura: Array 1D lineal

## 2.5.3.2.3.3.14.1: Ejemplo Completo: Mismo Dato en ROLAP vs MOLAP

Dato: Alice compró Laptop por \$1,000 en Enero

### ROLAP - Fact Table:

fact\_sales tabla:

| id | product_id | customer_id | amount | date       |
|----|------------|-------------|--------|------------|
| 1  | 101        | 201         | 1000   | 2024-01-15 |

← Este fact

↑  
FACT (valor)  
guardado en TABLA

Tipo estructura: Tabla relacional (2D)

Acceso: Scan + filtro

### MOLAP - Array Multidimensional:

Dimensiones (dictionaries):

Products: {101 → 0:"Laptop"}

Customers: {201 → 0:"Alice"}

Time: {Enero → 0}

Values Array:

|      |    |   |      |     |
|------|----|---|------|-----|
| 1000 | 20 | 0 | 1000 | ... |
|------|----|---|------|-----|

↑  
[0][0][0]  
FACT (valor)  
guardado en ARRAY

Tipo estructura: Array 1D

Acceso: Offset directo

## ¿Puede ser Array NxM? NO

Siempre es Array 1D, independiente de dimensiones del cubo

Cubo 3D (3×2×3):

→ Facts en array 1D de 18 valores: [v0, v1, v2, ..., v17]

Cubo 4D (10×5×3×7):

→ Facts en array 1D de 1,050 valores: [v0, v1, v2, ..., v1049]

Cubo 5D (4×3×2×6×5):

→ Facts en array 1D de 720 valores: [v0, v1, v2, ..., v719]

Los FACTS siempre en array 1D, nunca tabla 2D

## 2.5.3.2.3.3.14.2: Código C Simplificado

```
// Estructura MOLAP real

struct MOLAPCube {
 // Dimensiones
 Dictionary products; // "Laptop"→0, "Mouse"→1, ...
 Dictionary customers; // "Alice"→0, "Bob"→1
 Dictionary time; // "Jan"→0, "Feb"→1, "Mar"→2

 // Tamaños
 int dim_products = 3;
 int dim_customers = 2;
 int dim_time = 3;

 // ARRAY DE FACTS (valores) - 1D
 int64_t* facts; // Array 1D de 18 facts
 // NO es int64_t** (matriz)
 // ES int64_t* (vector 1D)

 // Total facts: 3 × 2 × 3 = 18
};

// Inicialización
cube.facts = malloc(18 * sizeof(int64_t));
// Crea array 1D con 18 facts: [f0, f1, f2, ..., f17]

// NO crea fact table:
// ✗ NO hay filas ni columnas
// ✗ NO hay estructura relacional

// Acceso a un fact
int64_t get_fact(int p, int c, int t) {
 int offset = (p * dim_customers * dim_time) +
 (c * dim_time) +
 t;
 return facts[offset]; // Lee fact del array 1D
}

// Ejemplo: get_fact(0, 0, 0)
// offset = 0
// return facts[0] = 1000 ← El fact
...
```

## Comparación Visual

NO es así (Fact Table - ROLAP):

 MOLAP NO usa esto:

Fact Table (estructura relacional):

| id | product | customer | amount |
|----|---------|----------|--------|
| 1  | 101     | 201      | 1000   |
| 2  | 102     | 201      | 20     |

← Filas

↑

Columns

Tabla 2D con filas y columnas

MOLAP NO tiene esta estructura

...

## Sí es así (Facts Array - MOLAP):

 MOLAP usa esto:

Facts Array (vector 1D):

|      |    |   |      |   |   |   |    |   |   |   |    |      |   |   |   |   |   |
|------|----|---|------|---|---|---|----|---|---|---|----|------|---|---|---|---|---|
| 1000 | 20 | 0 | 1000 | 0 | 0 | 0 | 20 | 0 | 0 | 0 | 50 | 1000 | 0 | 0 | 0 | 0 | 0 |
|------|----|---|------|---|---|---|----|---|---|---|----|------|---|---|---|---|---|

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

↑ fact

↑ fact

Vector 1D de facts

Sin filas ni columnas

### 2.5.3.2.3.3.14.3: Respuesta Directa + Resumen Final

#### 1. "¿Es vector columna o matriz NxM?"

 Ni uno ni otro

**Es un vector 1D PLANO de facts:**

[1000, 20, 0, 1000, 0, 0, 0, 20, 0, 0, 0, 0, 50, 1000, 0, 0, 0, 0, 0]  
fact fact fact

#### 2. "¿Cómo se ve ese array de facts?"

Memoria física (bytes):

Byte 0-7: [00 00 00 00 00 00 03 E8] ← fact: 1000  
Byte 8-15: [00 00 00 00 00 00 00 14] ← fact: 20  
Byte 16-23: [00 00 00 00 00 00 00 00] ← fact: 0  
Byte 24-31: [00 00 00 00 00 00 03 E8] ← fact: 1000

Array lineal de facts consecutivos en memoria

Sin estructura de tabla

#### 3. "¿Los facts están en fact tables?"

 Depende del sistema:

**ROLAP:** Facts EN fact\_sales (tabla relacional)

Estructura: Tabla 2D con filas/columnas

**MOLAP:** Facts EN values array (array 1D)

Estructura: Vector lineal

NO hay fact table

## Resumen Final

| Aspecto                | ROLAP (Relational OLAP)                                                | MOLAP (Multidimensional OLAP)                                                                                        |
|------------------------|------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| ¿Tiene facts?          | <input checked="" type="checkbox"/> Sí                                 | <input checked="" type="checkbox"/> Sí                                                                               |
| ¿Tiene fact table?     | <input checked="" type="checkbox"/> Sí (Tabla relacional)              |  No (Estructura multidimensional) |
| Estructura de facts    | Tabla 2D (Filas × Columnas)                                            | Array 1D (Lineal/Vector)                                                                                             |
| Almacenamiento         | Mezclado con dimensiones                                               | Separado de dimensiones                                                                                              |
| Tipo de dato           | Filas en tabla SQL                                                     | <code>int64_t values[N]</code> (C+ o similar)                                                                        |
| Acceso                 | Scan + Filtro (Select/Where)                                           | Offset directo (Cálculo de posición)                                                                                 |
| ¿Es matriz?            | <input checked="" type="checkbox"/> Sí (La tabla funciona como matriz) |  No (Es un vector indexado)       |
| ¿Tiene filas/columnas? | <input checked="" type="checkbox"/> Sí                                 |  No (Usa índices lógicos)         |

## Conclusión

**MOLAP SÍ tiene FACTS (valores), pero NO tiene FACT TABLES (tablas relacionales).**

Facts = Los valores numéricos (\$1000, 20 unidades, etc)

Fact Table = Estructura relacional con filas y columnas

MOLAP tiene lo primero, NO tiene lo segundo.

Los facts en MOLAP están en un array 1D multidimensional,  
NO en una tabla 2D relacional.

**El cubo N-dimensional LÓGICO se "aplana" a un array 1D FÍSICO que contiene todos los facts.**

## 2.5.3.2.3.3.15: Explicación ½ Hay 2 Estructuras Separadas

( )

### Lo Que Realmente Sucede en MOLAP

Hay DOS estructuras SEPARADAS:

#### 1. DIMENSION STORE (almacenamiento completo)

└ Guarda TODA la info de la dimensión

#### 2. FACTS ARRAY (array de valores)

└ Solo guarda los facts (números)

### 2.5.3.2.3.3.15.1 Estructura Real de MOLAP

#### 1. DIMENSION STORE - Almacena Toda la Información:

DIMENSION: Products (MOLAP storage)  
Almacenada en estructura multidimensional

Member ID: 0  
└ ProductKey: 101 (PK original)  
└ ProductName: "Laptop"  
└ Category: "Electronics"  
└ Brand: "Dell"  
└ Price: 1200.00  
└ SKU: "LAP-001"

Member ID: 1  
└ ProductKey: 102  
└ ProductName: "Mouse"  
└ Category: "Electronics"  
└ Brand: "Logitech"  
└ Price: 25.00  
└ SKU: "MOU-001"

Member ID: 2  
└ ... (Keyboard)

Esto SÍ guarda TODA la información  
 NO es un simple dictionary {0: "Laptop"}

#### 2. FACTS ARRAY - Solo Valores Numéricos:

FACTS ARRAY

Array[product\_id][customer\_id][time\_id]

[0][0][0] = 1000 ← Solo el valor (fact)  
[1][0][0] = 20  
[2][0][0] = 50

Solo números, NO información dimensional

## 2.5.3.2.3.3.15.2 Proceso Completo: ROLAP → MOLAP

### Paso 1: Datos Origen (ROLAP):

| DIMENSION TABLE: products |            |               |         |
|---------------------------|------------|---------------|---------|
| id                        | name       | category      | price   |
| 101                       | "Laptop"   | "Electronics" | 1200.00 |
| 102                       | "Mouse"    | "Electronics" | 25.00   |
| 103                       | "Keyboard" | "Electronics" | 50.00   |

| FACT TABLE: fact_sales |            |        |
|------------------------|------------|--------|
| id                     | product_id | amount |
| 1                      | 101        | 1000   |
| 2                      | 102        | 20     |
| 3                      | 103        | 50     |

### Paso 2: Construcción MOLAP:

| ETL PROCESS: |                                                                                                                                                                                                                                                                                                                                                                                  |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1.           | Procesar Dimension Products:                                                                                                                                                                                                                                                                                                                                                     |
|              | Leer: dimension table products completa                                                                                                                                                                                                                                                                                                                                          |
|              | Crear DIMENSION STORE en MOLAP:                                                                                                                                                                                                                                                                                                                                                  |
|              | <pre>Member 0:<br/>- All attributes from id=101<br/>- ProductKey: 101<br/>- Name: "Laptop"<br/>- Category: "Electronics"<br/>- Price: 1200.00<br/><br/>Member 1:<br/>- All attributes from id=102<br/>- Name: "Mouse"<br/>- Category: "Electronics"<br/>- Price: 25.00<br/><br/>Member 2:<br/>- All attributes from id=103<br/>- Name: "Keyboard"<br/>- ...<br/><br/>- ...</pre> |
|              | <input checked="" type="checkbox"/> TODA la información se copia a MOLAP<br><input checked="" type="checkbox"/> Se asignan IDs internos (0, 1, 2)                                                                                                                                                                                                                                |

2. Procesar Fact Table:

Leer: fact\_sales

Convertir FKs → Internal IDs:

- product\_id=101 → Buscar en dimension → Member ID=0
- product\_id=102 → Member ID=1
- product\_id=103 → Member ID=2

Almacenar en FACTS ARRAY:

Array[0][customer][time] = 1000

Array[1][customer][time] = 20

Array[2][customer][time] = 50

Solo facts (valores) en array

NO información dimensional aquí

## ⌚ Lo Que Me Confundió (y Te Confundí)

### Dictionary Encoding ≠ Dimension Store

LO QUE DIJE MAL:

"El Dictionary guarda toda la info"

LO CORRECTO:

Hay 3 cosas DIFERENTES:

#### 1. DIMENSION STORE (MOLAP)

- Almacena TODA la info dimensional
- ProductKey, Name, Category, Price, etc.
- Es una copia completa de la dimension table

#### 2. DICTIONARY ENCODING (técnica de compresión)

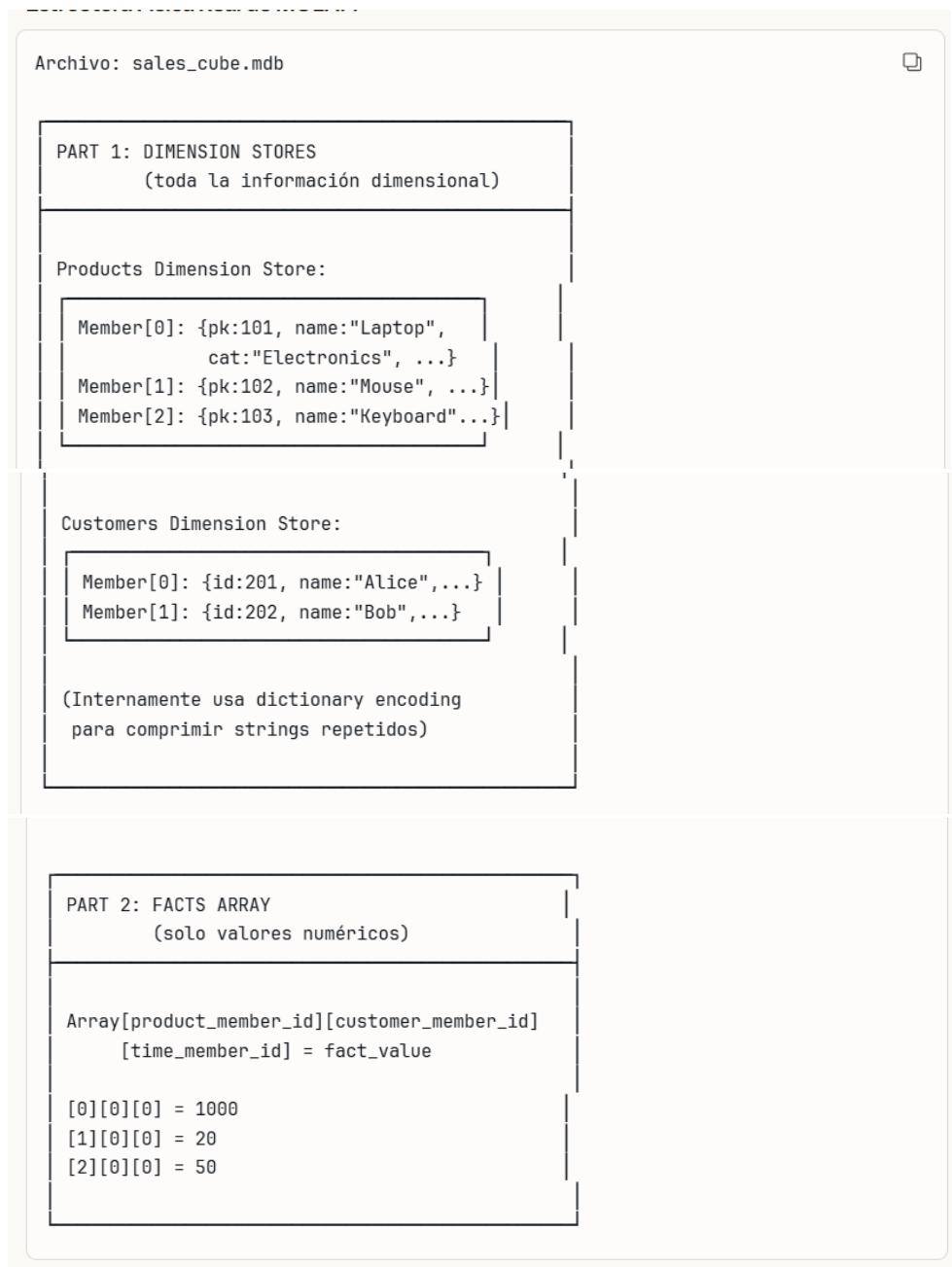
- Comprime strings repetidos internamente
- "Electronics" aparece 3 veces → guarda 1 vez
- Es una optimización INTERNA

#### 3. FACTS ARRAY

- Solo valores numéricos (facts)
- [1000, 20, 50, ...]

## 2.5.3.2.3.3.15.3: Visualización Correcta

### Estructura Física Real de MOLAP:



## 2.5.3.2.3.3.15.4: 🤝 Respuesta Directa a Tu Pregunta + Comparación Correcta

"¿El dictionary guarda solo {1: 'leche', 2: 'bebida'}?"

✗ NO - Eso fue mi error

Lo correcto:

El DIMENSION STORE (no dictionary simple) guarda:

```
{
 0: {
 'product_key': 101,
 'name': 'Laptop',
 'category': 'Electronics',
 'brand': 'Dell',
 'price': 1200.00,
 'sku': 'LAP-001',
 // ... todos los atributos
 },
 1: {
 'product_key': 102,
 'name': 'Mouse',
 'category': 'Electronics',
 'brand': 'Logitech',
 'price': 25.00,
 'sku': 'MOU-001'
 },
 // sku . . .
},
2: {
 'product_key': 103,
 'name': 'Keyboard',
 // ...
}
}
```

NO es un simple {0: "Laptop", 1: "Mouse"}

ES una estructura completa con TODOS los atributos

## ✓ Corrección de Mis Explicaciones Anteriores

Lo que DEBÍ haber dicho:

1. MOLAP tiene **DIMENSION STORES** que almacenan TODA la información dimensional (no solo strings)
2. **Dictionary encoding** es una técnica de COMPRESIÓN INTERNA que MOLAP usa para optimizar el almacenamiento de strings repetidos
3. El **FACTS ARRAY** usa IDs internos (0,1,2...) como índices para referenciar members en el dimension store

## 📊 Comparación Correcta

| Aspecto          | ROLAP                                              | MOLAP                                              |
|------------------|----------------------------------------------------|----------------------------------------------------|
| Dimensiones      | Dimension tables (PK, atributos)                   | Dimension stores (members + atributos)             |
| Facts            | Fact table con FKs                                 | Facts array con member IDs                         |
| Info dimensional | En dimension tables                                | COPIADA a dimension stores                         |
| Relación         | FK → PK                                            | Member ID (índice)                                 |
| Storage completo | <input checked="" type="checkbox"/> Sí (en tablas) | <input checked="" type="checkbox"/> Sí (en stores) |

## 🎯 Conclusión

Me disculpo por la confusión. La realidad es:

- MOLAP SÍ guarda TODA la información dimensional en structures llamadas "dimension stores"
- NO es un simple dictionary de strings
- Es una copia completa de las dimension tables con TODOS sus atributos
- Dictionary encoding es solo una optimización interna para comprimir
- El facts array usa member IDs (0,1,2...) como índices para referenciar estas dimensiones

#### 2.5.3.2.3.3.16: Explicación 2/2 Estructura Completa y Correcta - DIMENSION STORES

ACLARACIÓN CRÍTICA: Corrijo mi explicación anterior

---

#### 2.5.3.2.3.3.16.1 Estructura Real de MOLAP

MOLAP tiene DOS componentes principales separados:

1. DIMENSION STORES (almacenes de dimensiones)
  - └ Guardan TODA la información dimensional completa
2. FACTS ARRAY (array de hechos)
  - └ Guarda solo los valores numéricicos (facts)

#### 2.5.3.2.3.3.16.2 ¿Qué es un DIMENSION STORE?

DIMENSION STORE = Copia completa de la dimension table en formato multidimensional

En ROLAP tienes:

DIMENSION TABLE: products (tabla relacional)

| id  | name       | category  | price   | brand  |
|-----|------------|-----------|---------|--------|
| 101 | "Laptop"   | "Electr." | 1200.00 | "Dell" |
| 102 | "Mouse"    | "Electr." | 25.00   | "Logi" |
| 103 | "Keyboard" | "Electr." | 50.00   | "MS"   |

Estructura: Tabla 2D (filas × columnas)

Tipo: Relacional



Proceso ETL (copia)



En MOLAP tienes:

DIMENSION STORE: products (estructura MD)

Member ID: 0  
  └ ProductKey: 101  
  └ ProductName: "Laptop"  
  └ Category: "Electronics"  
  └ Price: 1200.00  
  └ Brand: "Dell"

Member ID: 1  
  └ ProductKey: 102  
  └ ProductName: "Mouse"  
  └ Category: "Electronics"  
  └ Price: 25.00  
  └ Brand: "Logitech"

Member ID: 2  
  └ ProductKey: 103  
  └ ProductName: "Keyboard"  
  └ ...

Estructura: Multidimensional comprimida

Tipo: MOLAP propio

- TODA la información se copia
- CADA atributo se guarda
- Se asignan Member IDs (0, 1, 2...)

## 2.5.3.2.3.3.16.3 Estructura Física Completa en MOLAP

### SECCIÓN 1: METADATA (información del cubo)

- Nombre del cubo: "Sales Analysis"
- Número de dimensiones: 3
- Número de medidas: 2
- Fecha de Última actualización
- Nivel de agregación

Tamaño: ~1 KB

### SECCIÓN 2: DIMENSION STORES (almacenes dimensionales)

#### PRODUCTS DIMENSION STORE

Member 0:

```
ProductKey: 101 (int)
ProductName: "Laptop" (string)
Category: "Electronics" (string)
SubCategory: "Computers" (string)
Brand: "Dell" (string)
Price: 1200.00 (float)
Cost: 800.00 (float)
LaunchDate: 2023-01-15 (date)
Active: true (boolean)
SKU: "LAP-001" (string)
```

Member 1:

```
ProductKey: 102 (int)
ProductName: "Mouse" (string)
Category: "Electronics" (string)
SubCategory: "Peripherals" (string)
Brand: "Logitech" (string)
Price: 25.00 (float)
Cost: 10.00 (float)
LaunchDate: 2023-03-20 (date)
Active: true (boolean)
SKU: "MOU-001" (string)
```

Member 2:

```
ProductKey: 103 (int)
ProductName: "Keyboard" (string)
Category: "Electronics" (string)
... (todos los atributos)
```

Tamaño: ~50 KB

| CUSTOMERS DIMENSION STORE                                                                                                                                                                                                                                                                                                                                                                                                                    |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Member 0:                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <pre>CustomerKey: 201 (int) FirstName: "Alice" (string) LastName: "Johnson" (string) Email: "alice@company.com" (string) Phone: "+1-555-0100" (string) BirthDate: 1989-05-15 (date) Gender: "F" (string) MaritalStatus: "Married" (string) Occupation: "Engineer" (string) YearlyIncome: 75000.00 (float) City: "Seattle" (string) State: "WA" (string) Country: "USA" (string) RegisterDate: 2020-03-15 (date) Active: true (boolean)</pre> |

Member 1:

|                                                                                                                  |
|------------------------------------------------------------------------------------------------------------------|
| <pre>CustomerKey: 202 (int) FirstName: "Bob" (string) LastName: "Smith" (string) ... (todos los atributos)</pre> |
|------------------------------------------------------------------------------------------------------------------|

Tamaño: ~80 KB

#### TIME DIMENSION STORE

Member 0:

|                                                                                                                                                                                                                                                                                                                                   |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>DateKey: 20240115 (int) FullDate: 2024-01-15 (date) Year: 2024 (int) Quarter: 1 (int) Month: 1 (int) MonthName: "January" (string) Day: 15 (int) DayOfWeek: 1 (int) DayName: "Monday" (string) WeekOfYear: 3 (int) IsWeekend: false (boolean) IsHoliday: false (boolean) FiscalYear: 2024 (int) FiscalQuarter: 3 (int)</pre> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Member 1:

|                                   |
|-----------------------------------|
| <pre>  DateKey: 20240116...</pre> |
|-----------------------------------|

Member 2:

|                                   |
|-----------------------------------|
| <pre>  DateKey: 20240210...</pre> |
|-----------------------------------|

|                                   |
|-----------------------------------|
| <pre>  DateKey: 20240210...</pre> |
|-----------------------------------|

Tamaño: ~120 KB

TOTAL DIMENSION STORES: ~250 KB

SECCIÓN 3: FACTS ARRAY  
(solo valores numéricicos)

```
Array[product_member][customer_member]
[time_member] = {sales_amount, quantity}
```

Posición 0: [1000.00, 1] ← [0][0][0]

Posición 1: [ 20.00, 1] ← [1][0][0]

Posición 2: [ 50.00, 1] ← [2][0][0]

Posición 3: [1000.00, 1] ← [0][1][0]

...

Solo números (facts), NO info dimensional

Tamaño: ~150 KB (18 celdas × 2 medidas × 8B)

SECCIÓN 4: AGGREGATIONS (pre-calculadas)  
(opcional - para velocidad)

Agregaciones por producto:

```
Array[product_member][ALL][ALL] = totales
```

Agregaciones por cliente:

```
Array[ALL][customer_member][ALL] = totales
```

Gran total:

```
Array[ALL][ALL][ALL] = gran_total
```

Tamaño: ~50 KB

TOTAL ARCHIVO .mdb: ~450 KB

## 2.5.3.2.3.3.16.4 Zoom: Dentro de un DIMENSION STORE

### Products Dimension Store - Vista Detallada:

Archivo físico: products\_dimension.dat

**HEADER (metadatos del store)**

- Nombre: "Products"
- Total members: 3
- Total atributos: 10
- Compresión: Dictionary encoding habilitada
- Última actualización: 2024-01-29

**ATTRIBUTE DEFINITIONS (esquema)**

Atributo 0: ProductKey (Type: INT32)  
Atributo 1: ProductName (Type: STRING)  
Atributo 2: Category (Type: STRING)  
Atributo 3: SubCategory (Type: STRING)  
Atributo 4: Brand (Type: STRING)  
Atributo 5: Price (Type: FLOAT64)  
Atributo 6: Cost (Type: FLOAT64)  
Atributo 7: LaunchDate (Type: DATE)  
Atributo 8: Active (Type: BOOLEAN)  
Atributo 9: SKU (Type: STRING)

**STRING DICTIONARY (compresión interna)**  
Para optimizar strings repetidos

StringID 0: "Electronics"  
StringID 1: "Computers"  
StringID 2: "Peripherals"  
StringID 3: "Dell"  
StringID 4: "Logitech"  
StringID 5: "Microsoft"  
StringID 6: "Laptop"  
StringID 7: "Mouse"  
StringID 8: "Keyboard"  
StringID 9: "LAP-001"  
StringID 10: "MOU-001"  
StringID 11: "KEY-001"

### MEMBERS DATA (datos reales)

#### Member 0

```
ProductKey: 101
ProductName: →StringID 6 ("Laptop")
Category: →StringID 0 ("Electronics")
SubCategory: →StringID 1 ("Computers")
Brand: →StringID 3 ("Dell")
Price: 1200.00
Cost: 800.00
LaunchDate: 2023-01-15
Active: true
SKU: →StringID 9 ("LAP-001")
```

Bytes: 0-80

#### Member 1

```
ProductKey: 102
ProductName: →StringID 7 ("Mouse")
Category: →StringID 0 ("Electronics")
SubCategory: →StringID 2 ("Peripherals")
Brand: →StringID 4 ("Logitech")
Price: 25.00
Cost: 10.00
LaunchDate: 2023-03-20
Active: true
SKU: →StringID 10 ("MOU-001")
```

Bytes: 81-160

#### Member 2

```
ProductKey: 103
ProductName: →StringID 8 ("Keyboard")
Category: →StringID 0 ("Electronics")
SubCategory: →StringID 2 ("Peripherals")
Brand: →StringID 5 ("Microsoft")
Price: 50.00
Cost: 20.00
LaunchDate: 2023-02-10
Active: true
SKU: →StringID 11 ("KEY-001")
```

Bytes: 161-240

### MEMBER INDEX (para búsqueda rápida)

```
ProductKey 101 → Member 0
ProductKey 102 → Member 1
ProductKey 103 → Member 2
```

Para traducir FK → Member ID

## Notas importantes:

1. **String Dictionary:** "Electronics" aparece 3 veces, pero se guarda UNA vez y se referencia con StringID 0
2. **Compression ratio:** ~60% gracias a dictionary encoding
3. **Access time:** Lectura directa de member por ID = O(1)

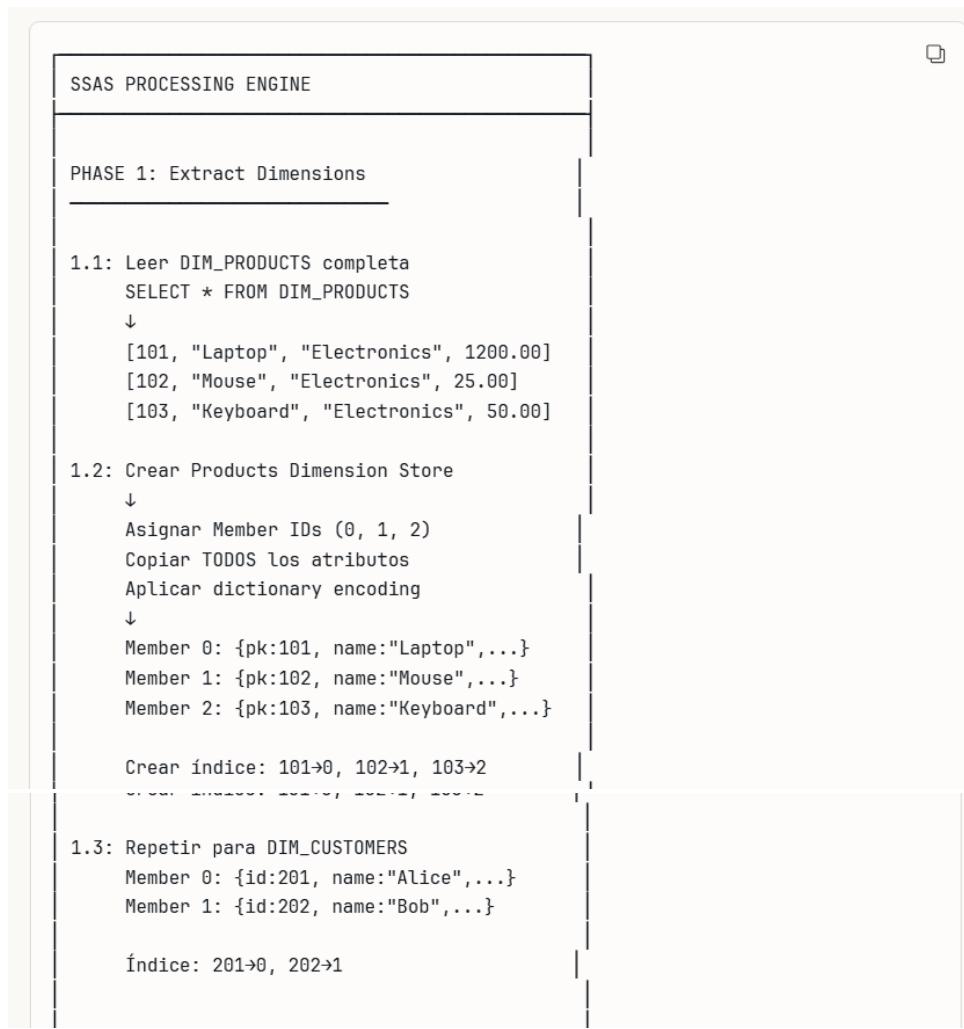
## 2.5.3.2.3.3.16.5 ↗

### Ejemplo Real: Construcción del Cubo

#### Paso 1: Datos Origen en DWH (ROLAP):

| DATA WAREHOUSE (SQL Server/Oracle)                                                                                                                                                                                                                                                                                                            |            |             |         |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|-------------|---------|----|------------|-------------|--------|---------|-----------|-----------|---------|------------|---------|-----------|-------|-----|------------|-----------|-------|
| DIM_PRODUCTS:                                                                                                                                                                                                                                                                                                                                 |            |             |         |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
| <table border="1"><thead><tr><th>id</th><th>name</th><th>category</th><th>price</th></tr></thead><tbody><tr><td>101</td><td>"Laptop"</td><td>"Electr."</td><td>1200.00</td></tr><tr><td>102</td><td>"Mouse"</td><td>"Electr."</td><td>25.00</td></tr><tr><td>103</td><td>"Keyboard"</td><td>"Electr."</td><td>50.00</td></tr></tbody></table> |            |             |         | id | name       | category    | price  | 101     | "Laptop"  | "Electr." | 1200.00 | 102        | "Mouse" | "Electr." | 25.00 | 103 | "Keyboard" | "Electr." | 50.00 |
| id                                                                                                                                                                                                                                                                                                                                            | name       | category    | price   |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
| 101                                                                                                                                                                                                                                                                                                                                           | "Laptop"   | "Electr."   | 1200.00 |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
| 102                                                                                                                                                                                                                                                                                                                                           | "Mouse"    | "Electr."   | 25.00   |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
| 103                                                                                                                                                                                                                                                                                                                                           | "Keyboard" | "Electr."   | 50.00   |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
| Tamaño: 500 bytes (tabla relacional)                                                                                                                                                                                                                                                                                                          |            |             |         |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
| DIM_CUSTOMERS:                                                                                                                                                                                                                                                                                                                                |            |             |         |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
| <table border="1"><thead><tr><th>id</th><th>name</th><th>city</th></tr></thead><tbody><tr><td>201</td><td>"Alice"</td><td>"Seattle"</td></tr><tr><td>202</td><td>"Bob"</td><td>"Portland"</td></tr></tbody></table>                                                                                                                           |            |             |         | id | name       | city        | 201    | "Alice" | "Seattle" | 202       | "Bob"   | "Portland" |         |           |       |     |            |           |       |
| id                                                                                                                                                                                                                                                                                                                                            | name       | city        |         |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
| 201                                                                                                                                                                                                                                                                                                                                           | "Alice"    | "Seattle"   |         |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
| 202                                                                                                                                                                                                                                                                                                                                           | "Bob"      | "Portland"  |         |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
| Tamaño: 200 bytes                                                                                                                                                                                                                                                                                                                             |            |             |         |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
| FACT_SALES:                                                                                                                                                                                                                                                                                                                                   |            |             |         |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
| <table border="1"><thead><tr><th>id</th><th>product_id</th><th>customer_id</th><th>amount</th></tr></thead><tbody><tr><td>1</td><td>101</td><td>201</td><td>1000</td></tr><tr><td>2</td><td>102</td><td>201</td><td>20</td></tr><tr><td>3</td><td>103</td><td>202</td><td>50</td></tr></tbody></table>                                        |            |             |         | id | product_id | customer_id | amount | 1       | 101       | 201       | 1000    | 2          | 102     | 201       | 20    | 3   | 103        | 202       | 50    |
| id                                                                                                                                                                                                                                                                                                                                            | product_id | customer_id | amount  |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
| 1                                                                                                                                                                                                                                                                                                                                             | 101        | 201         | 1000    |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
| 2                                                                                                                                                                                                                                                                                                                                             | 102        | 201         | 20      |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
| 3                                                                                                                                                                                                                                                                                                                                             | 103        | 202         | 50      |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
| Tamaño: 400 bytes                                                                                                                                                                                                                                                                                                                             |            |             |         |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |
| TOTAL DWH: ~1,100 bytes                                                                                                                                                                                                                                                                                                                       |            |             |         |    |            |             |        |         |           |           |         |            |         |           |       |     |            |           |       |

## Paso 2: Proceso ETL (Extract-Transform-Load):



## PHASE 2: Process Facts

### 2.1: Leer FACT\_SALES

```
SELECT product_id, customer_id, amount
 FROM FACT_SALES
↓
[101, 201, 1000]
[102, 201, 20]
[103, 202, 50]
```

### 2.2: Traducir FKs → Member IDs

```
Fila 1: [101, 201, 1000]
↓
product_id=101 → buscar en índice → 0
customer_id=201 → buscar en índice → 0
↓
Coordenadas: [0][0][0] = 1000

Fila 2: [102, 201, 20]
↓
Coordenadas: [1][0][0] = 20

Fila 3: [103, 202, 50]
↓
Coordenadas: [2][1][0] = 50
```

.....

### 2.3: Escribir Facts Array

```
Array[0][0][0] = 1000
Array[1][0][0] = 20
Array[2][1][0] = 50
```

### PHASE 3: Build Aggregations (opcional)

---

#### 3.1: Calcular totales por producto

Array[0][ALL][ALL] = 1000 (Laptop)  
Array[1][ALL][ALL] = 20 (Mouse)  
Array[2][ALL][ALL] = 50 (Keyboard)

#### 3.2: Calcular totales por cliente

Array[ALL][0][ALL] = 1020 (Alice)  
Array[ALL][1][ALL] = 50 (Bob)

#### 3.3: Gran total

Array[ALL][ALL][ALL] = 1070

### PHASE 4: Write to Disk

---

Escribir sales\_cube.mdb:

- Metadata
- Products Dimension Store
- Customers Dimension Store
- Time Dimension Store
- Facts Array
- Aggregations

Cubo MOLAP creado



### RESULTADO:

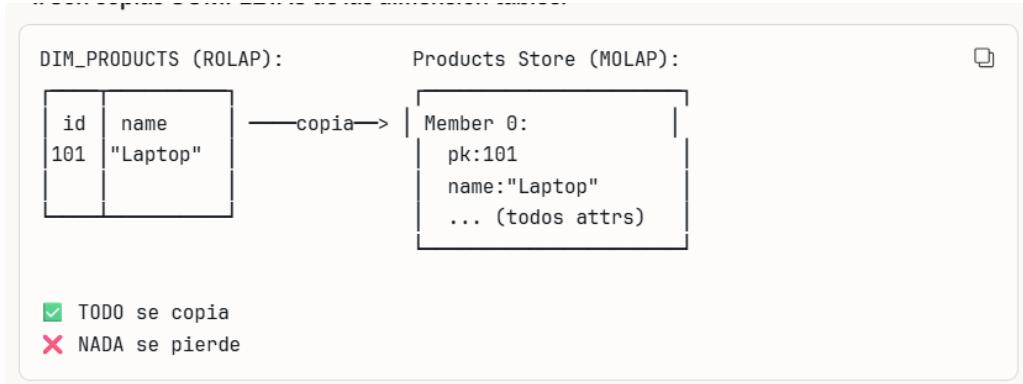
- Dimension Stores: 250 KB
- Facts Array: 150 KB
- Aggregations: 50 KB
- TOTAL: ~450 KB

vs DWH original: ~1,100 bytes

Ratio de expansión: 450 KB / 1.1 KB = ~400x  
(Por agregaciones pre-calculadas)

## 2.5.3.2.3.3.16.6 🎉 Características Clave de DIMENSION STORES

### 1. Son copias COMPLETAS de las dimension tables:



### 2. Usan Member IDs en lugar de PKs:

ROLAP:  
product\_id=101 (PK) → producto específico

MOLAP:  
ProductKey=101 guardado en Member 0  
Member ID=0 usado en Facts Array

Ventaja: IDs consecutivos (0,1,2...) = 100% eficiencia

### 3. Soportan múltiples tipos de datos:

Products Dimension Store puede tener:

- Integers: ProductKey, YearLaunched
- Strings: Name, Category, Brand
- Floats: Price, Cost, Weight
- Dates: LaunchDate, DiscontinuedDate
- Booleans: Active, OnSale
- Decimals: TaxRate, Discount

No limitado a strings  
 Guarda tipos originales

### 4. Usan compresión interna (Dictionary Encoding):

Sin compresión:  
Member 0: Category="Electronics" (12 bytes)  
Member 1: Category="Electronics" (12 bytes)  
Member 2: Category="Electronics" (12 bytes)  
Total: 36 bytes

Con dictionary encoding:  
String Dictionary: ID 0 = "Electronics" (12 bytes)

Member 0: Category=>StringID 0 (1 byte)  
Member 1: Category=>StringID 0 (1 byte)  
Member 2: Category=>StringID 0 (1 byte)  
Total: 15 bytes (58% ahorro)

## **5. Mantienen índices para búsqueda rápida:**

Products Dimension Store tiene:



1. Member Index (PK → Member ID):

101 → 0

102 → 1

103 → 2

2. Attribute Indexes (para búsqueda):

Brand="Dell" → [Member 0]

Brand="Logitech" → [Member 1]

Category="Electronics" → [Members 0,1,2]

Búsqueda O(1) en lugar de O(n)

## 2.5.3.2.3.3.16.7 Cómo Trabajan Juntos: DIMENSION STORES + FACTS ARRAY

Query: "Ventas de Laptop para Alice"

PASO 1: Usuario hace query

```
SELECT SUM(Amount)
FROM Sales
WHERE Product = 'Laptop'
AND Customer = 'Alice'
```



PASO 2: MOLAP Engine traduce a Member IDs

Buscar "Laptop" en Products Store:

- └ Attribute Index: Name="Laptop"
- └ Member ID = 0

Buscar "Alice" en Customers Store:

- └ Attribute Index: Name="Alice"
- └ Member ID = 0

Resultado: Necesito Array[0][0][ALL]



PASO 3: Leer Facts Array

Facts Array:

```
Array[0][0][0] = 1000 ← Enero
Array[0][0][1] = 0 ← Febrero
Array[0][0][2] = 1000 ← Marzo
```

SUM = 1000 + 0 + 1000 = 2000

Tiempo: 0.01 segundos



PASO 4: Retornar resultado

\$2,000

(Sin necesidad de JOIN ni cálculo)



## Query con Atributos: "Ventas de productos Dell en Seattle"

PASO 1: Query

```
SELECT SUM(Amount)
FROM Sales
WHERE Product.Brand = 'Dell'
AND Customer.City = 'Seattle'
```

↓

PASO 2: Buscar en Dimension Stores

Products Store - buscar Brand='Dell':

```
Member 0: Brand="Dell" ✓
Member 1: Brand="Logitech" ✗
Member 2: Brand="Microsoft" ✗
```

Resultado: Member IDs = [0]

Customers Store - buscar City='Seattle':

```
Member 0: City="Seattle" ✓
Member 1: City="Portland" ✗
```

Resultado: Member IDs = [0]

Coordenadas: Array[0][0][ALL]

↓

PASO 3: Leer Facts

```
SUM(Array[0][0][0], Array[0][0][1], ...)
= $2,000
```

**Nota:** Los atributos (Brand, City) NO están en el Facts Array, están en los Dimension Stores.

## 2.5.3.2.3.3.16.8 ⚡ Resumen: DIMENSION STORES vs Dimension Tables + Conclusión Final

| Aspecto       | ROLAP Dimension Table         | MOLAP Dimension Store                                           |
|---------------|-------------------------------|-----------------------------------------------------------------|
| Estructura    | Tabla relacional 2D           | Estructura multidimensional                                     |
| Formato       | Filas x Columnas              | Members con atributos                                           |
| Identificador | PK (101, 102, 103)            | Member ID (0, 1, 2)                                             |
| Contenido     | Todos los atributos           | <input checked="" type="checkbox"/> Todos los atributos (COPIA) |
| Relaciones    | FK → PK                       | Member ID (índice)                                              |
| Compresión    | Baja (sin)                    | Alta (dictionary encoding)                                      |
| Índices       | B-Tree en PK                  | Multiple indexes + Member ID                                    |
| Ubicación     | Base de datos relacional      | Archivo MOLAP (.mdb)                                            |
| Acceso        | JOIN necesario                | Lookup directo                                                  |
| Tipos datos   | String, int, float, date, etc | String, int, float, date, etc                                   |
| Actualización | UPDATE directo                | Requiere reprocesar cubo                                        |

### ✓ Conclusión Final

#### DIMENSION STORES en MOLAP:

1.  Son **copias COMPLETAS** de las dimension tables originales
2.  Guardan **TODOS los atributos** (no solo strings)
3.  Usan **Member IDs** (0,1,2...) en lugar de PKs para optimizar
4.  Aplican **dictionary encoding** internamente para comprimir
5.  Mantienen **índices múltiples** para búsqueda rápida
6.  Se separan del **Facts Array** que solo tiene valores numéricos
7.  Permiten queries por **cualquier atributo** sin JOIN

#### Facts Array:

- Solo guarda valores numéricos (facts)
- Usa Member IDs como índices
- No contiene información dimensional

#### Juntos forman MOLAP:

- Dimension Stores = Info dimensional completa
- Facts Array = Valores numéricos
- Result = Acceso ultra rápido sin JOINs

## 2.5.3.2.3.4: Preguntas sobre MOLAP | Aclaracion de dudas

### 2.5.3.2.3.4.1: Un cubo real tiene MULTIPLES MEDIDAS (measures/facts)

#### El Problema Real

Un cubo típico tiene múltiples medidas:

Coordenada [1][1][1] (Mouse-Bob-Feb) puede tener:

- Sales Amount: \$20
- Quantity Sold: 1 unidad
- Cost: \$10
- Profit: \$10
- Discount: 10%
- Tax: \$2
- Average Price: \$20
- Max Transaction: \$20
- Min Transaction: \$20
- Transaction Count: 1

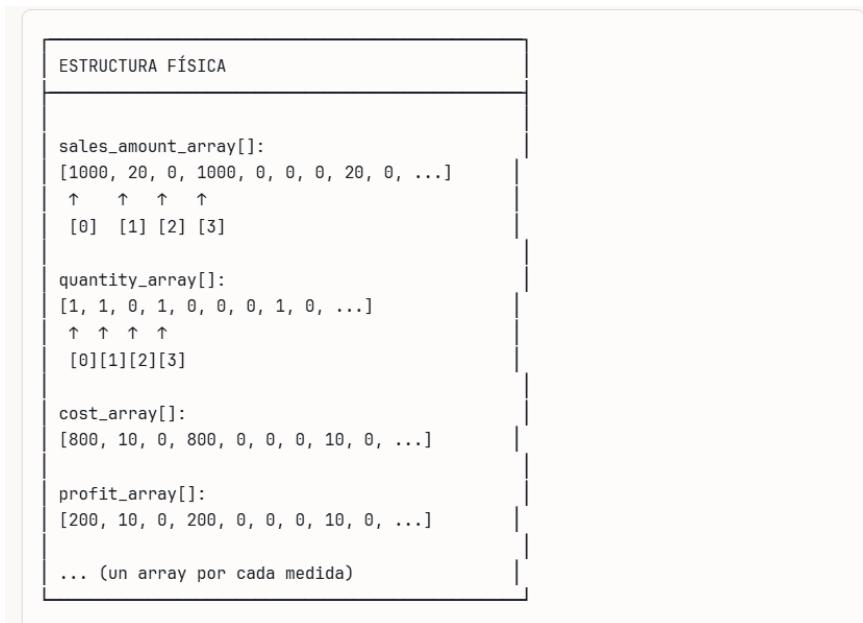
¿Cómo se guarda esto en el array 1D físico?

## 2.5.3.2.3.4.1.1: 📈 Soluciones Reales en MOLAP (3 enfoques principales)

Hay 3 enfoques principales:

### ENFOQUE 1: Arrays Separados por Medida (MÁS COMÚN)

Un array 1D por cada medida:



VENTAJAS:

- ✓ Simple de implementar
- ✓ Cada array puede tener tipo diferente (int64, float64)
- ✓ Fácil agregar/quitar medidas
- ✓ Cache-friendly (acceso secuencial por medida)

DESVENTAJAS:

- ✗ Múltiples arrays que mantener
- ✗ Para obtener todos los valores de [1][1][1] necesitas leer de múltiples arrays

---

### Ejemplo Concreto:

The screenshot shows a code editor window with the following content:

```
Coordenadas [1][1][1] (Mouse-Bob-Feb):
offset = (1 * 2 * 3) + (1 * 3) + 1 = 10

Leer todas las medidas:
- sales_amount_array[10] = 0
- quantity_array[10] = 0
- cost_array[10] = 0
- profit_array[10] = 0
- discount_array[10] = 0

Total: 5 lecturas separadas
```

---

### ENFOQUE 2: Array Intercalado (INTERLEAVED)

## Todas las medidas juntas, secuencialmente:

The screenshot shows a code editor window with a title bar 'ESTRUCTURA FÍSICA (intercalada)'. The main area contains the following code and annotations:

```
measures_array[]:
Coord [0][0][0]:
[1000, 1, 800, 200, 0, ← 5 medidas juntas
↑ ↑ ↑ ↑ ↑
sales qty cost prof disc

Coord [1][0][0]:
20, 1, 10, 10, 10, ← 5 medidas juntas

Coord [2][0][0]:
0, 0, 0, 0, 0, ← 5 medidas juntas

Coord [0][1][0]:
1000, 1, 800, 200, 0, ← 5 medidas juntas

...]

Patrón: [m1, m2, m3, m4, m5] repetido
```

### ESTRUCTURA:

Array lineal con medidas intercaladas:

```
[sales0, qty0, cost0, profit0, disc0,
sales1, qty1, cost1, profit1, disc1,
sales2, qty2, cost2, profit2, disc2, ...]
```

### VENTAJAS:

- Todas las medidas de una coord juntas (locality)
- Un solo array que mantener
- Leer coord completa = 1 operación

### DESVENTAJAS:

- Acceso a una sola medida requiere saltos
- Todas las medidas deben ser mismo tipo
- Agregar medida = reestructurar todo

---

### Ejemplo Concreto:

```

Coordenadas [1][1][1] (Mouse-Bob-Feb):
Tiene 5 medidas

offset_base = (1 * 2 * 3) + (1 * 3) + 1 = 10
offset_con_medidas = offset_base * num_measures
 = 10 * 5 = 50

Leer todas las medidas de [1][1][1]:
measures_array[50] = 0 ← sales
measures_array[51] = 0 ← quantity
measures_array[52] = 0 ← cost
measures_array[53] = 0 ← profit
measures_array[54] = 0 ← discount

Total: 1 lectura contigua (5 valores consecutivos)

```

## ¿Qué significa offset\_con\_medidas = 10 × 5 = 50?

```

offset_base = 10
↓
Este es el offset de la COORDENADA [1][1][1]
(la posición en el array de facts)

num_measures = 5
↓
Hay 5 medidas por cada coordenada:
- sales
- quantity
- cost
- profit
- discount

offset_con_medidas = 10 × 5 = 50
↓
Para leer LAS 5 MEDIDAS de esa coordenada,
empiezas en la posición 50 del measures_array

```

## Visualización

```

measures_array: [0,0,0,0,0, ..., 0,0,0,0,0, ...]
 ↑ ↑
 pos 0-4 pos 50-54
 coord[0][0][0] coord[1][1][1]

measures_array[50] = sales de [1][1][1]
measures_array[51] = quantity de [1][1][1]
measures_array[52] = cost de [1][1][1]
measures_array[53] = profit de [1][1][1]
measures_array[54] = discount de [1][1][1]

```

## En Resumen

**Multiplicas por 5 porque cada coordenada ocupa 5 espacios consecutivos en el array (una posición por medida).**

Es como decir: "La coordenada #10 empieza en la posición 50 del array, porque cada coordenada usa 5 slots."

## 🎯 Diferencia entre offset 10 y offset 50

Son ARRAYS DIFERENTES

ARRAY 1: facts\_array (coordenadas)

Position 0 → coordenada [0][0][0]  
Position 1 → coordenada [0][0][1]  
Position 2 → coordenada [0][0][2]  
...  
Position 10 → coordenada [1][1][1] ← AQUÍ  
...

Este array NO guarda medidas  
Solo dice "qué coordenada es el #10"

ARRAY 2: measures\_array (valores de medidas)

Position 0-4 → 5 medidas de coordenada #0  
Position 5-9 → 5 medidas de coordenada #1  
Position 10-14 → 5 medidas de coordenada #2  
...  
Position 50-54 → 5 medidas de coordenada #10 ← AQUÍ |  
...

Estructura:  
[50] = sales de coord #10  
[51] = quantity de coord #10  
[52] = cost de coord #10  
[53] = profit de coord #10  
[54] = discount de coord #10

### Resumen Ultra Breve

offset 10 → "Esta es la coordenada número 10"  
(en el espacio conceptual del cubo)

offset 50 → "Los valores de la coordenada #10  
empiezan en la posición 50"  
(en el array físico de medidas)

**10 es el índice de la coordenada**

**50 es dónde empiezan sus valores en memoria**

Por eso: **10 × 5 medidas = 50**

## ENFOQUE 3: Medida como Dimensión Extra (MENOS COMÚN)

Tratar medidas como otra dimensión:

## ESTRUCTURA LÓGICA

Array[product][customer][time][measure]  
↑  
4ta dimensión

### Measures Dimension:

- Member 0: "Sales Amount"
- Member 1: "Quantity"
- Member 2: "Cost"
- Member 3: "Profit"
- Member 4: "Discount"

## ESTRUCTURA FÍSICA:

Array 1D de 4 dimensiones

Cubo original:  $3 \times 2 \times 3 = 18$  coordenadas

Con medidas:  $3 \times 2 \times 3 \times 5 = 90$  celdas

### Array físico:

[val0, val1, val2, ..., val89]

### Mapeo:

- [0][0][0] → offset 0 (Laptop-Alice-Jan-Sales)
  - [0][0][0][1] → offset 1 (Laptop-Alice-Jan-Quantity)
  - [0][0][0][2] → offset 2 (Laptop-Alice-Jan-Cost)
  - [0][0][0][3] → offset 3 (Laptop-Alice-Jan-Profit)
  - [0][0][0][4] → offset 4 (Laptop-Alice-Jan-Discount)
  - [1][0][0] → offset 5 (Mouse-Alice-Jan-Sales)
- ...

## VENTAJAS:

- Consistente con el modelo dimensional
- Fórmula offset funciona igual
- Queries pueden filtrar por medida

## DESVENTAJAS:

- Explota el tamaño del cubo ( $\times \text{num\_measures}$ )
- Menos intuitivo conceptualmente
- Medidas no son realmente una dimensión

---

## Ejemplo Concreto:

Coordenadas [1][1][1][2] (Mouse-Bob-Feb-Cost):



Cubo: 3 products × 2 customers × 3 times × 5 measures

```
offset = (1 * 2 * 3 * 5) +
 (1 * 3 * 5) +
 (1 * 5) +
 2
= 30 + 15 + 5 + 2
= 52
```

```
array[52] = 0 (cost de Mouse-Bob-Feb)
```

#### 2.5.3.2.3.4.1.2: Comparación: Los 3 Enfoques

Tabla Comparativa:

| Aspecto          | Arrays Separados      | Intercalado           | Medida=Dimensión      |
|------------------|-----------------------|-----------------------|-----------------------|
| Estructura       | N arrays 1D           | 1 array 1D            | 1 array 1D (4D)       |
| Tamaño           | $18 \times N$ medidas | $18 \times N$ medidas | $18 \times N$ medidas |
| Localidad coord  | ✗ Baja                | ✓ Alta                | ✓ Alta                |
| Localidad medida | ✓ Alta                | ✗ Baja                | ✗ Baja                |
| Agregar medida   | ✓ Fácil               | ✗ Reestructura        | ✗ Reestructura        |
| Tipos mixtos     | ✓ Sí                  | ✗ No                  | ✗ No                  |
| Complejidad      | 🟡 Media               | 🟢 Baja                | 🔴 Alta                |
| Usado por        | SSAS, Essbase         | Algunos MOLAP         | Raro                  |

### 2.5.3.2.3.4.1.3: ⚡ Lo Más Común en la Práctica

Microsoft SQL Server Analysis Services (SSAS):



Cada medida en su propio archivo

Mismo tamaño: 18 valores cada uno

Mismo orden de coordenadas

### 2.5.3.2.3.4.1.4: 📈 Visualización Completa: Tu Ejemplo | Los 2 enfoques

Cubo con 5 medidas:

## Coordenadas 3D → Valores múltiples



[0][0][0] (Laptop-Alice-Jan):

- └ Sales: \$1000
- └ Quantity: 1
- └ Cost: \$800
- └ Profit: \$200
- └ Discount: 0%

[1][0][0] (Mouse-Alice-Jan):

- └ Sales: \$20
- └ Quantity: 1
- └ Cost: \$10
- └ Profit: \$10
- └ Discount: 10%

[2][0][0] (Keyboard-Alice-Jan):

- └ Sales: \$0
- └ Quantity: 0
- └ Cost: \$0
- └ Profit: \$0
- └ Discount: 0%



**Enfoque 1: Arrays Separados (COMÚN):**

### Enfoque 1: Arrays Separados (COMUN):

SALES\_AMOUNT.DAT (array 1D)

```
[1000, 20, 0, 1000, 0, 0, 0, 20, 0, 0, 0,
50, 1000, 0, 0, 0, 0, 0]
↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
0 1 2 3 4 5 6 7 8 9 10
```

QUANTITY.DAT (array 1D)

```
[1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0,
1, 1, 0, 0, 0, 0, 0]
```

COST.DAT (array 1D)

```
[800, 10, 0, 800, 0, 0, 0, 10, 0, 0, 0,
20, 800, 0, 0, 0, 0, 0]
```

PROFIT.DAT (array 1D)

```
[200, 10, 0, 200, 0, 0, 0, 10, 0, 0, 0,
30, 200, 0, 0, 0, 0, 0]
```

DISCOUNT.DAT (array 1D)

```
[0, 10, 0, 0, 0, 0, 0, 10, 0, 0, 0,
5, 0, 0, 0, 0, 0, 0]
```

Query: Todas las medidas de [1][0][0] (Mouse-Alice-Jan)  
offset = 1

Leer:

- sales\_amount[1] = 20
- quantity[1] = 1
- cost[1] = 10
- profit[1] = 10
- discount[1] = 10

Total: 5 lecturas (una de cada array)

### Enfoque 2: Intercalado:

MEASURES.DAT (array 1D intercalado)

|                        |                   |
|------------------------|-------------------|
| [1000, 1, 800, 200, 0, | ← [0][0][0] todas |
| 20, 1, 10, 10, 10,     | ← [1][0][0] todas |
| 0, 0, 0, 0, 0,         | ← [2][0][0] todas |
| 1000, 1, 800, 200, 0,  | ← [0][1][0] todas |
| 0, 0, 0, 0, 0,         | ← [1][1][0] todas |
| 0, 0, 0, 0, 0,         | ← [2][1][0] todas |
| ...]                   |                   |

Índices: 0 1 2 3 4 5 6 7 8 9...

Query: Todas las medidas de [1][0][0] (Mouse-Alice-Jan)

offset\_base = 1

offset\_start = 1 \* 5 = 5

Leer:

measures[5] = 20 (sales)

measures[6] = 1 (quantity)

measures[7] = 10 (cost)

measures[8] = 10 (profit)

measures[9] = 10 (discount)

Total: 1 lectura contigua (5 valores consecutivos)

## 2.5.3.2.3.4.1.5: Respuesta a la pregunta + Conclusión

Sí, tienes razón:

Coordenada [1][1][1] NO guarda solo 1 valor



Guarda MÚLTIPLES valores (medidas):

```
[1][1][1] = {
 sales: valor1,
 quantity: valor2,
 cost: valor3,
 profit: valor4,
 discount: valor5
}
```

En la práctica (enfoque más común):

NO es: array\_1d[position] = [val1, val2, val3, val4, val5]



Sí es:

```
sales_array[position] = val1
quantity_array[position] = val2
cost_array[position] = val3
profit_array[position] = val4
discount_array[position] = val5
```

Múltiples arrays 1D paralelos

Mismo índice = misma coordenada

### Conclusión

Tu intuición es correcta: Una coordenada tiene múltiples valores.

Implementación real: La mayoría de sistemas MOLAP usan **arrays separados** por medida, no un array de arrays.

Por qué:

- Más flexible
- Tipos diferentes por medida
- Agregar/quitar medidas fácil
- Cache-friendly para queries específicas

Tu ejemplo [1][1][1] = [valor, valor, valor, valor] es conceptualmente correcto, pero físicamente se implementa como múltiples arrays paralelos con el mismo índice.

## 2.5.3.2.3.4.2: Estructura Real de MOLAP sin confusiones

### 2.5.3.2.3.4.2.1: MOLAP: Estructura de Almacenamiento Definitiva

#### ⌚ Resumen Ejecutivo

MOLAP almacena datos en DOS tipos de estructuras completamente separadas:

1. **Dimension Stores** (.dim files) = Diccionarios con atributos de dimensiones
2. **Fact Arrays** (.dat files) = Arrays numéricos con valores de medidas

## 1. Estructura de Archivos en Disco

```
/cube_data/
 |
 +-- dimensions/ # Dimension Stores
 |
 +-- products.dim # Archivo 1: Dimensión Productos
 +-- customers.dim # Archivo 2: Dimensión Clientes
 +-- time.dim # Archivo 3: Dimensión Tiempo

 +-- facts/ # Facts Arrays
 +-- sales_amount.dat # Array 1: Ventas
 +-- quantity.dat # Array 2: Cantidad
 +-- cost.dat # Array 3: Costos
```

Cada dimensión = 1 archivo .dim

Cada medida = 1 archivo .dat

## 2. DIMENSION STORES: Estructura Interna

Concepto: Members Array

"Members Array" = Tabla de filas dentro de un archivo .dim

Analogía SQL:

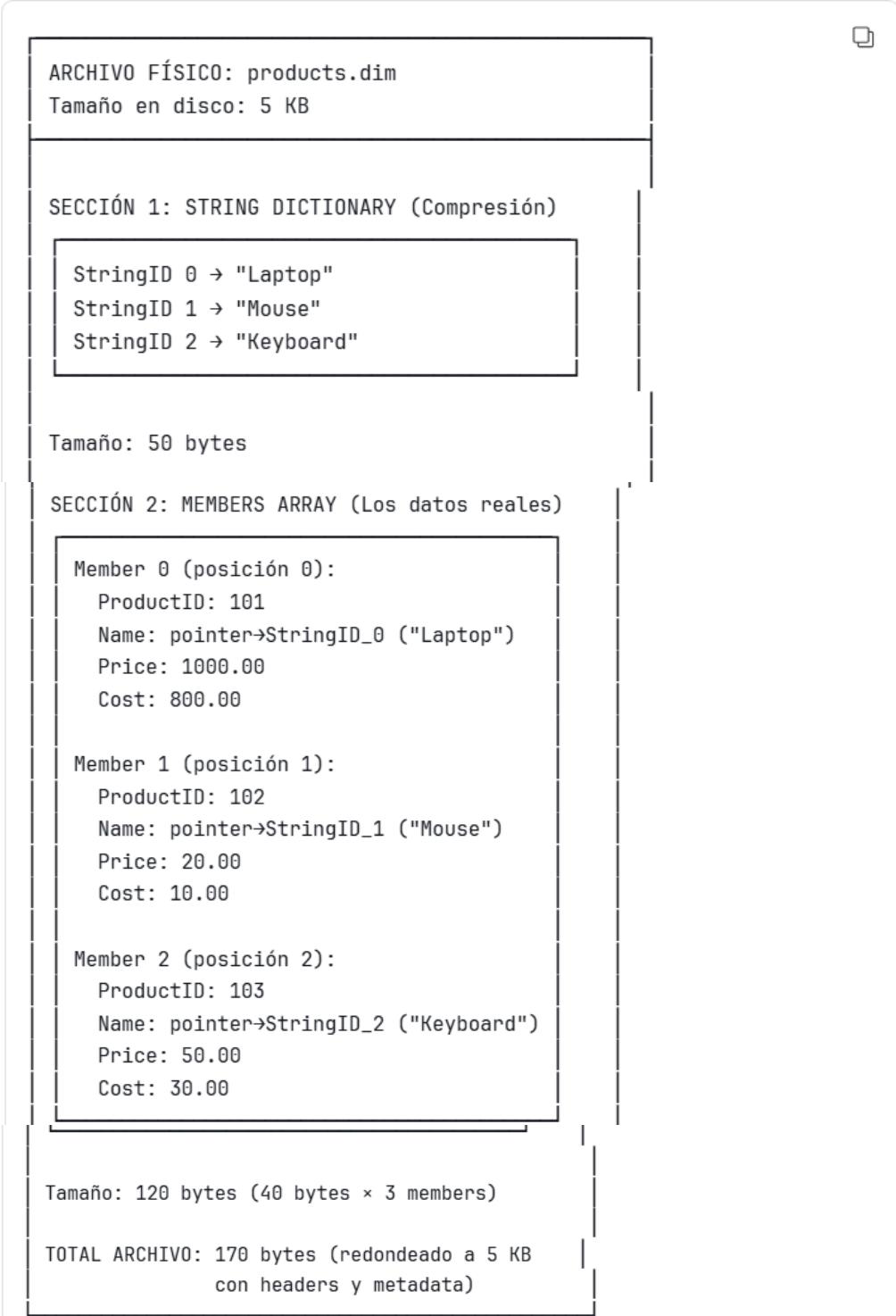
```
SELECT * FROM dim_products;
```

| ID | Name     | Price | Cost |            |
|----|----------|-------|------|------------|
| 0  | Laptop   | 1000  | 800  | ← Member 0 |
| 1  | Mouse    | 20    | 10   | ← Member 1 |
| 2  | Keyboard | 50    | 30   | ← Member 2 |

En MOLAP, esto se guarda como:

- String dictionary: ["Laptop", "Mouse", "Keyboard"]
- Members array: Array de structs con los datos

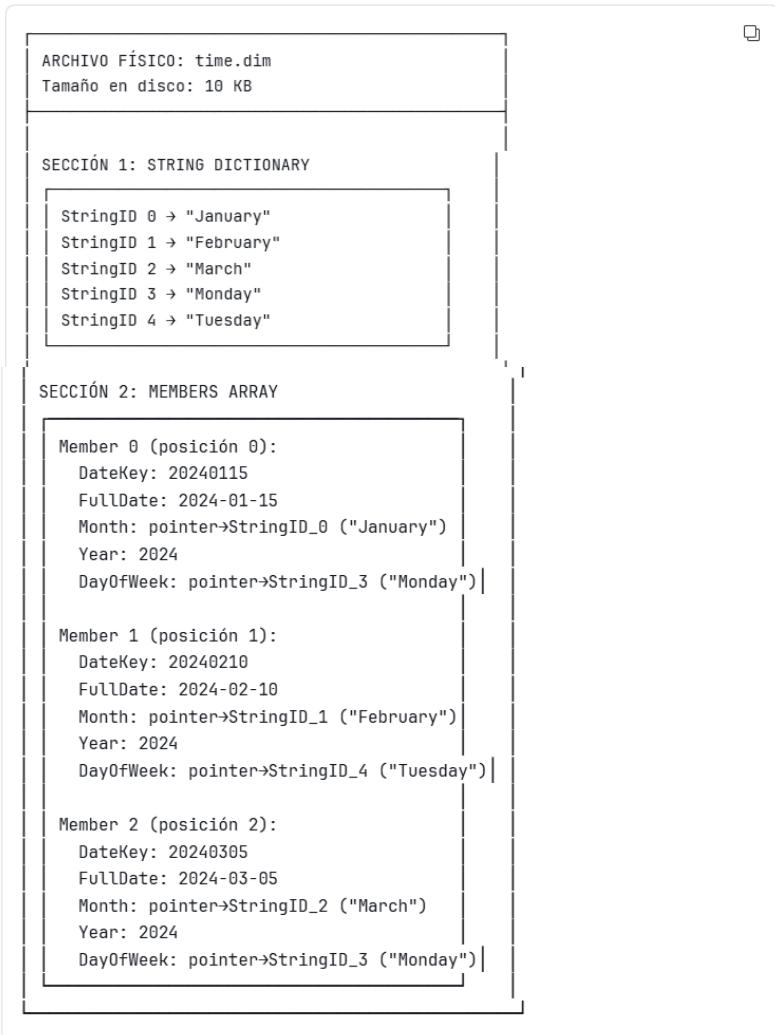
## Ejemplo Real: products.dim



## Representación en Código:

```
Members array en memoria (conceptual)
products_members = [
 {
 'position': 0,
 'product_id': 101,
 'name_ptr': 0, # apunta a "Laptop"
 'price': 1000.00,
 'cost': 800.00
 },
 {
 'position': 1,
 'product_id': 102,
 'name_ptr': 1, # apunta a "Mouse"
 'price': 20.00,
 'cost': 10.00
 },
 {
 'position': 2,
 'product_id': 103,
 'name_ptr': 2, # apunta a "Keyboard"
 'price': 50.00,
 'cost': 30.00
 }
]
```

## Ejemplo Real: time.dim



## Representación en Código:

```
Members array
time_members = [
 {
 'position': 0,
 'date_key': 20240115,
 'full_date': datetime(2024, 1, 15),
 'month_ptr': 0, # "January"
 'year': 2024,
 'day_of_week_ptr': 3 # "Monday"
 },
 {
 'position': 1,
 'date_key': 20240210,
 'full_date': datetime(2024, 2, 10),
 'month_ptr': 1, # "February"
 'year': 2024,
 'day_of_week_ptr': 4 # "Tuesday"
 },
 {
 'position': 2,
 'date_key': 20240305,
 'full_date': datetime(2024, 3, 5),
 'month_ptr': 2, # "March"
 'year': 2024,
 'day_of_week_ptr': 3 # "Monday"
 }
]
```

## 3. FACTS ARRAYS: Almacenamiento de Medidas

### Estructura Física

Los facts arrays NO contienen nombres, solo valores numéricos.

```
ARCHIVO FÍSICO: sales_amount.dat
Tamaño en disco: 144 bytes

Array 1D de FLOAT64 (8 bytes por valor)

[Bytes 0-7]: 1000.00 (posición 0)
[Bytes 8-15]: 20.00 (posición 1)
[Bytes 16-23]: 1000.00 (posición 2)
[Bytes 24-31]: 0.00 (posición 3)
[Bytes 32-39]: 0.00 (posición 4)
[Bytes 40-47]: 0.00 (posición 5)
[Bytes 48-55]: 0.00 (posición 6)
[Bytes 56-63]: 20.00 (posición 7)
[Bytes 64-71]: 0.00 (posición 8)
[Bytes 72-79]: 0.00 (posición 9)
[Bytes 80-87]: 0.00 (posición 10)
[Bytes 88-95]: 50.00 (posición 11)
[Bytes 96-103]: 1000.00 (posición 12)
[Bytes 104-111]: 0.00 (posición 13)
[Bytes 112-119]: 0.00 (posición 14)
[Bytes 120-127]: 0.00 (posición 15)
[Bytes 128-135]: 0.00 (posición 16)
[Bytes 136-143]: 0.00 (posición 17)

TOTAL: 18 posiciones × 8 bytes = 144 bytes
```

## Representación Visual (Solo Valores):

sales\_amount.dat:

|      |    |      |   |   |   |   |    |   |   |    |    |      |    |    |    |    |
|------|----|------|---|---|---|---|----|---|---|----|----|------|----|----|----|----|
| 0    | 1  | 2    | 3 | 4 | 5 | 6 | 7  | 8 | 9 | 10 | 11 | 12   | 13 | 14 | 15 | 16 |
| 17   |    |      |   |   |   |   |    |   |   |    |    |      |    |    |    |    |
| 1000 | 20 | 1000 | 0 | 0 | 0 | 0 | 20 | 0 | 0 | 0  | 50 | 1000 | 0  | 0  | 0  | 0  |
| 0    |    |      |   |   |   |   |    |   |   |    |    |      |    |    |    |    |

quantity.dat:

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  |

cost.dat:

|     |    |     |   |   |   |   |    |   |   |    |    |     |    |    |    |    |    |
|-----|----|-----|---|---|---|---|----|---|---|----|----|-----|----|----|----|----|----|
| 0   | 1  | 2   | 3 | 4 | 5 | 6 | 7  | 8 | 9 | 10 | 11 | 12  | 13 | 14 | 15 | 16 | 17 |
| 800 | 10 | 800 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 0  | 30 | 800 | 0  | 0  | 0  | 0  | 0  |

### ¡IMPORTANTE!

Los nombres (Laptop, Alice, January) NO están en estos arrays.  
Solo hay números puros.

## 4. Mapeo: Coordenadas → Posición en Array

Fórmula del Offset

Para cubo de 3 dimensiones [Products][Customers][Time]:

```
offset = product_id × (num_customers × num_time)
+ customer_id × num_time
+ time_id
```

Dimensiones del cubo:

- Products: 3 (IDs 0, 1, 2)
- Customers: 2 (IDs 0, 1)
- Time: 3 (IDs 0, 1, 2)

Total posiciones:  $3 \times 2 \times 3 = 18$

## Tabla de Mapeo Completa

| Coordenadas | Cálculo Offset<br>$0 \times 6 + 0 \times 3 + 0 = 0$ | Qué Representa<br>Laptop - Alice - January<br>sales_amount[0] = 1000 |
|-------------|-----------------------------------------------------|----------------------------------------------------------------------|
| [1][0][0]   | $1 \times 6 + 0 \times 3 + 0 = 6$                   | Mouse - Alice - January<br>sales_amount[6] = 0                       |
| [2][0][0]   | $2 \times 6 + 0 \times 3 + 0 = 12$                  | Keyboard - Alice - January<br>sales_amount[12] = 1000                |
| [0][1][0]   | $0 \times 6 + 1 \times 3 + 0 = 3$                   | Laptop - Bob - January<br>sales_amount[3] = 0                        |
| [1][1][0]   | $1 \times 6 + 1 \times 3 + 0 = 9$                   | Mouse - Bob - January<br>sales_amount[9] = 0                         |
| [2][1][0]   | $2 \times 6 + 1 \times 3 + 0 = 15$                  | Keyboard - Bob - January<br>sales_amount[15] = 0                     |
| [0][0][1]   | $0 \times 6 + 0 \times 3 + 1 = 1$                   | Laptop - Alice - February<br>sales_amount[1] = 20                    |
| [1][0][1]   | $1 \times 6 + 0 \times 3 + 1 = 7$                   | Mouse - Alice - February<br>sales_amount[7] = 20                     |
| [2][0][1]   | $2 \times 6 + 0 \times 3 + 1 = 13$                  | Keyboard - Alice - February<br>sales_amount[13] = 0                  |
| [0][1][1]   | $0 \times 6 + 1 \times 3 + 1 = 4$                   | Laptop - Bob - February<br>sales_amount[4] = 0                       |
| [1][1][1]   | $1 \times 6 + 1 \times 3 + 1 = 10$                  | Mouse - Bob - February<br>sales_amount[10] = 0                       |
| [2][1][1]   | $2 \times 6 + 1 \times 3 + 1 = 16$                  | Keyboard - Bob - February<br>sales_amount[16] = 0                    |
| [0][0][2]   | $0 \times 6 + 0 \times 3 + 2 = 2$                   | Laptop - Alice - March<br>sales_amount[2] = 1000                     |
| [1][0][2]   | $1 \times 6 + 0 \times 3 + 2 = 8$                   | Mouse - Alice - March<br>sales_amount[8] = 0                         |
| [2][0][2]   | $2 \times 6 + 0 \times 3 + 2 = 14$                  | Keyboard - Alice - March<br>sales_amount[14] = 0                     |

|           |                                    |                                                |
|-----------|------------------------------------|------------------------------------------------|
| [0][1][2] | $0 \times 6 + 1 \times 3 + 2 = 5$  | Laptop - Bob - March<br>sales_amount[5] = 0    |
| [1][1][2] | $1 \times 6 + 1 \times 3 + 2 = 11$ | Mouse - Bob - March<br>sales_amount[11] = 50   |
| [2][1][2] | $2 \times 6 + 1 \times 3 + 2 = 17$ | Keyboard - Bob - March<br>sales_amount[17] = 0 |

## 2.5.3.2.3.4.2.2: Proceso Completo: Query Real

Query: "¿Cuánto vendió Alice de Laptops en January?"

PASO 1: Buscar en products.dim

Abrir archivo: products.dim  
Buscar: "Laptop"

Resultado:  
Member 0: Name="Laptop"  
Member ID = 0

PASO 2: Buscar en customers.dim

Abrir archivo: customers.dim  
Buscar: "Alice"

Resultado:  
Member 0: Name="Alice"  
Member ID = 0

PASO 3: Buscar en time.dim

Abrir archivo: time.dim  
Buscar: Month="January"

Resultado:  
Member 0: Month="January"  
Member ID = 0

PASO 4: Calcular offset en facts array

Coordenadas: [0][0][0]

Fórmula:  
offset = 0×(2×3) + 0×3 + 0  
= 0×6 + 0 + 0  
= 0

Posición en array: 0

PASO 5: Leer valor en sales\_amount.dat

Abrir archivo: sales\_amount.dat  
Leer posición: 0

Bytes 0-7: 1000.00

RESULTADO FINAL: \$1000

## 2.5.3.2.3.4.2.3: Resumen Visual Completo

DIMENSION STORES (archivos .dim)  
= Diccionarios con atributos

products.dim:

```
Members Array (tabla de productos):
[
 {pos:0, name:"Laptop", price:1000, cost:800},
 {pos:1, name:"Mouse", price:20, cost:10},
 {pos:2, name:"Keyboard", price:50, cost:30}
]
```

customers.dim:

```
Members Array (tabla de clientes):
[
 {pos:0, name:"Alice", city:"Seattle"},
 {pos:1, name:"Bob", city:"Portland"}
]
```

time.dim:

```
Members Array (tabla de fechas):
[
 {pos:0, month:"January", year:2024},
 {pos:1, month:"February", year:2024},
 {pos:2, month:"March", year:2024}
]
```

FACTS ARRAYS (archivos .dat)  
= Solo valores numéricos

```
sales_amount.dat:
[1000, 20, 1000, 0, 0, 0, 0, 20, 0, 0, 0, 50, 1000, ...]

quantity.dat:
[1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, ...]

cost.dat:
[800, 10, 800, 0, 0, 0, 0, 10, 0, 0, 0, 30, 800, ...]
```

CONEXIÓN: Members → Facts

Para obtener valor:

1. Nombre → Member ID (dimension stores)  
"Laptop" → ID 0  
"Alice" → ID 0  
"January" → ID 0
2. Member IDs → Offset (fórmula matemática)  
[0][0][0] → offset 0
3. Offset → Valor (facts array)  
sales\_amount[0] = 1000

RESPUESTA: \$1000

## 2.5.3.2.3.4.2.4: Conceptos Clave Finales

### 1. Members Array

"Members Array" = Array de structs con datos de dimensión



NO es un concepto separado.

ES la forma de representar una tabla dentro de un .dim file.

Igual que en SQL:

- Una tabla tiene filas
- En MOLAP, las filas se llaman "members"
- Se guardan en un array de structs

### 2. Separación de Estructuras

Dimension Stores (.dim):



- └ Guardan: Nombres, atributos, metadata
- └ Formato: Structs con pointers a strings
- └ Ejemplo: {name:"Laptop", price:1000}

Facts Arrays (.dat):



- └ Guardan: Solo valores numéricicos
- └ Formato: Array puro de FLOAT64/INT32
- └ Ejemplo: [1000, 20, 0, 1000, ...]

### 3. El Mapeo

Member IDs → Coordenadas → Offset → Valor



"Laptop" (string en .dim)

    ↓ búsqueda

Member ID 0

    ↓ forma coordenada [0][0][0]

    ↓ cálculo

Offset 0

    ↓ lectura

sales\_amount[0] = 1000 (número en .dat)

## 2.5.3.2.3.4.2.5: Conclusión

MOLAP = Dos tipos de archivos separados:

1. Dimension Stores (.dim) = "Diccionarios" con members arrays (tablas de atributos)
2. Facts Arrays (.dat) = Arrays numéricicos puros

El "members array" es simplemente la representación en array de una tabla dimensional.

No hay magia. Es una tabla SQL guardada de forma optimizada con compresión de strings y acceso rápido por índice.

## 2.5.3.2.3.4.3.3: MOLAP: Preguntas Avanzadas - Guía Completa

### 2.5.3.2.3.4.3.1: VALORES DUPLICADOS: ¿Cómo se Diferencian?

Pregunta: Si tengo ventas de \$1000 en múltiples coordenadas, ¿cómo sé cuál es cuál?

Respuesta: Por su POSICIÓN (offset) en el array

EJEMPLO REAL:



Facts Array (Sales Amount):

```
[1000, 20, 0, 1000, 0, 0, 0, 20, 0, 0, 0, 50, 1000, 0, 0, 0, 0, 0]
↑ ↑ ↑
pos 0 pos 3 pos 12
```

Tres valores de \$1000, pero en POSICIONES diferentes:

Posición 0 → offset 0 → coordenadas [0][0][0]  
= Laptop-Alice-Jan = \$1000

Posición 3 → offset 3 → coordenadas [0][1][0]  
= Laptop-Bob-Jan = \$1000

Posición 12 → offset 12 → coordenadas [0][0][2]  
= Laptop-Alice-Mar = \$1000

- MISMO VALOR (\$1000)
- DIFERENTES COORDENADAS
- DIFERENTES POSICIONES (offset)

## La Posición (Offset) ES la Clave de Diferenciación

NO SE GUARDA ASÍ (MAL):

```
{
 value: 1000,
 product: "Laptop",
 customer: "Alice",
 time: "Jan"
}
```

✗ Esto sería como ROLAP (tabla relacional)

SE GUARDA ASÍ (CORRECTO):

```
Array posición 0 = 1000
Array posición 3 = 1000
Array posición 12 = 1000
```

- El offset IMPLICA las coordenadas
- No necesitas guardar las coordenadas
- offset → fórmula → coordenadas

## Ejemplo Práctico: Diferenciar Valores Duplicados

Query: "¿Cuánto vendí de Laptop en Marzo?"



PASO 1: Traducir a coordenadas

- Product: "Laptop" → Member ID 0
- Time: "March" → Member ID 2
- Customer: ALL (todas las posiciones)

PASO 2: Calcular offsets necesarios

- [0][0][2] → offset 12 (Laptop-Alice-Mar)
- [0][1][2] → offset 15 (Laptop-Bob-Mar)

PASO 3: Leer valores

- sales\_array[12] = 1000
- sales\_array[15] = 0

PASO 4: Sumar

- Total = 1000 + 0 = 1000

- Aunque hay múltiples \$1000 en el array,  
el OFFSET me dice cuál corresponde a cada coordenada

## 2.5.3.2.3.4.3.2: 12 34 2. DIMENSIONES con INT y FLOAT (no solo strings)

Pregunta: ¿Qué pasa con dimensiones que tienen números?

Respuesta: Dimension Stores guardan TODOS los tipos

The screenshot shows a software interface for managing a 'TIME DIMENSION STORE'. The title bar indicates it's a 'TIME DIMENSION STORE (Ejemplo con múltiples tipos)'. Below the title, there are two sections: 'Member 0:' and 'Member 1:'.  
  
Under 'Member 0:', there is a large box containing various data fields with their types:

- DateKey: 20240115 (int32)
- FullDate: 2024-01-15 (date)
- Year: 2024 (int16)
- Quarter: 1 (int8)
- Month: 1 (int8)
- MonthName: "January" (string)
- Day: 15 (int8)
- DayOfWeek: 1 (int8)
- DayName: "Monday" (string)
- WeekOfYear: 3 (int8)
- IsWeekend: false (boolean)
- IsHoliday: false (boolean)
- FiscalYear: 2024 (int16)
- FiscalQuarter: 3 (int8)
- Temperature: 5.5 (float32) °C
- ExchangeRate: 1.08 (float64) USD/EUR

A small downward arrow points from the last field, 'ExchangeRate', to another line of text: 'ExchangeRate: 1.08 (float64) USD/EUR'.  
  
Under 'Member 1:', there are three listed fields:

- DateKey: 20240116...
- Year: 2024...
- Temperature: 6.2 (float32)

- Strings: MonthName, DayName
- Integers: Year, Month, Day, Quarter
- Floats: Temperature, ExchangeRate
- Dates: FullDate
- Booleans: IsWeekend, IsHoliday

TODOS los tipos se guardan en el Dimension Store

---

## Estructura en Memoria para Tipos Mixtos

| MEMBER 0 (en bytes)        |                                 |
|----------------------------|---------------------------------|
| Bytes 0-3:                 | DateKey (int32) = 20240115      |
| Bytes 4-11:                | FullDate (int64) = timestamp    |
| Bytes 12-13:               | Year (int16) = 2024             |
| Byte 14:                   | Quarter (int8) = 1              |
| Byte 15:                   | Month (int8) = 1                |
| Bytes 16-23:               | MonthName (pointer) → "January" |
| Byte 24:                   | Day (int8) = 15                 |
| Byte 25:                   | DayOfWeek (int8) = 1            |
| Bytes 26-33:               | DayName (pointer) → "Monday"    |
| Byte 34:                   | WeekOfYear (int8) = 3           |
| Byte 35:                   | IsWeekend (bool) = 0            |
| Byte 36:                   | IsHoliday (bool) = 0            |
| Bytes 37-38:               | FiscalYear (int16) = 2024       |
| Byte 39:                   | FiscalQuarter (int8) = 3        |
| Bytes 40-43:               | Temperature (float32) = 5.5     |
| Bytes 44-51:               | ExchangeRate (float64) = 1.08   |
| TOTAL: 52 bytes por member |                                 |

Cada tipo ocupa su espacio correspondiente:

- int8: 1 byte
- int16: 2 bytes
- int32: 4 bytes
- float32: 4 bytes
- float64: 8 bytes
- string: 8 bytes (pointer al string dictionary)
- boolean: 1 byte

## Queries con Atributos Numéricos

Query: "Ventas en días con temperatura > 10°C"

PASO 1: Buscar en Time Dimension Store

- Filtrar: Temperature > 10
- Resultado: Member IDs [5, 8, 12, 15, ...]

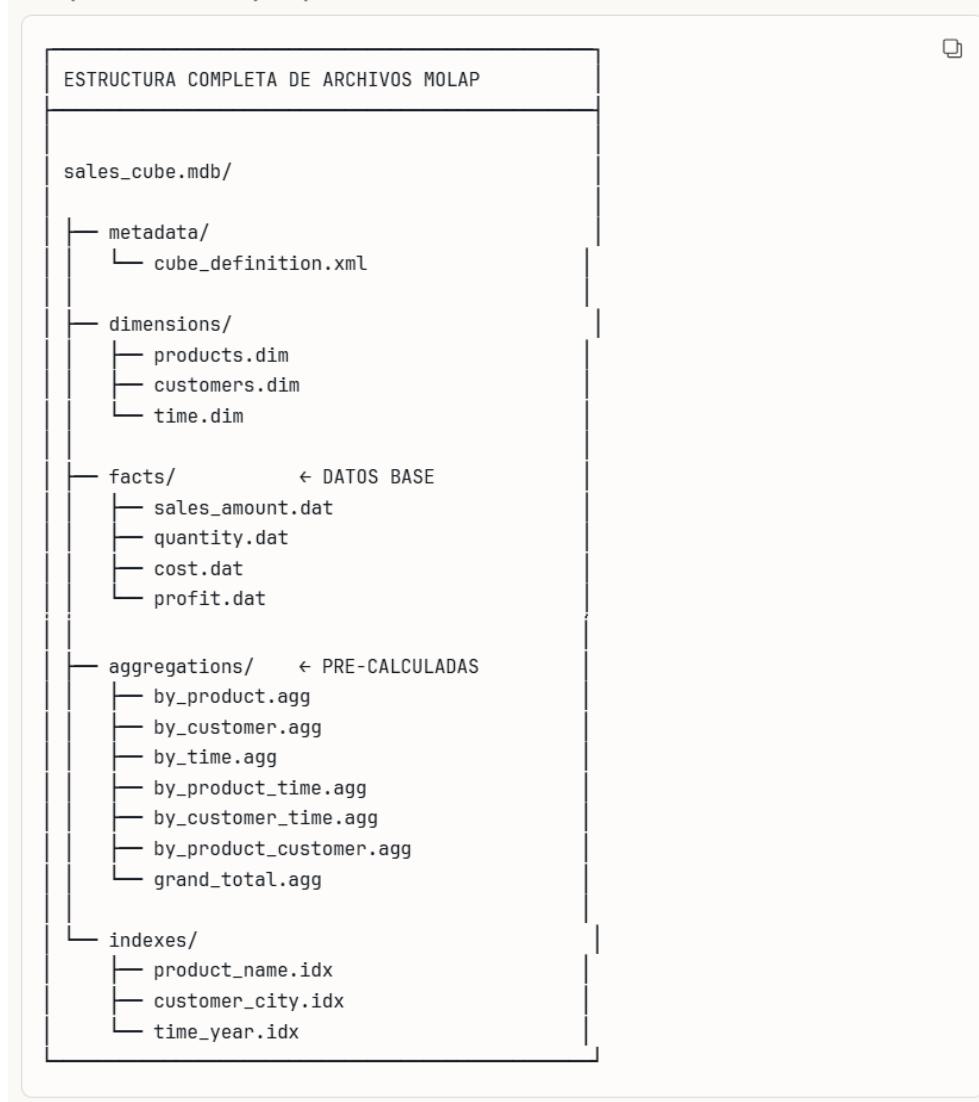
PASO 2: Para cada Member ID encontrado

- Calcular offset correspondiente
- Leer valor del facts array
- Sumar

- Puedes filtrar por CUALQUIER atributo  
(string, int, float, boolean, date)

## 2.5.3.2.3.4.3.3: AGREGACIONES: ¿Se Guardan en Arrays Aparte?

Respuesta: Sí, en arrays separados



## Estructura de Agregaciones (Detallada)

FACTS (granularidad base):

Array 3D: [3 products][2 customers][3 months] = 18 celdas



facts/sales\_amount.dat

```
[1000, 20, 0, 1000, 0, 0,
 0, 20, 0, 0, 0, 50,
 1000, 0, 0, 0, 0, 0]
```

Tamaño: 18 celdas × 8 bytes = 144 bytes

AGGREGATION: by\_product.agg

Array 1D: [3 products] = 3 celdas

aggregations/by\_product.agg

```
[2000, 40, 50]
↑ ↑ ↑
Laptop Mouse Keyboard
(todas las combos de customer+time)
```

Tamaño: 3 celdas × 8 bytes = 24 bytes

AGGREGATION: by\_customer.agg

Array 1D: [2 customers] = 2 celdas

aggregations/by\_customer.agg

```
[1020, 50]
↑ ↑
Alice Bob
(todas las combos de product+time)
```

Tamaño: 2 celdas × 8 bytes = 16 bytes

AGGREGATION: by\_time.agg

Array 1D: [3 months] = 3 celdas

aggregations/by\_time.agg

[2020, 20, 50]  
↑    ↑    ↑  
Jan   Feb Mar  
(todas las combos de product+customer)

Tamaño: 3 celdas × 8 bytes = 24 bytes

AGGREGATION: by\_product\_time.agg

Array 2D: [3 products][3 months] = 9 celdas

aggregations/by\_product\_time.agg

[2000, 20, 1000, ← Laptop (Jan, Feb, Mar)  
40, 0, 0,       ← Mouse  
0, 0, 50]       ← Keyboard

Tamaño: 9 celdas × 8 bytes = 72 bytes



AGGREGATION: grand\_total.agg

Array 0D: [] = 1 celda

aggregations/grand\_total.agg

[2070] ← Total de TODO

Tamaño: 1 celda × 8 bytes = 8 bytes

TOTAL AGREGACIONES: ~144 bytes

TOTAL FACTS: 144 bytes

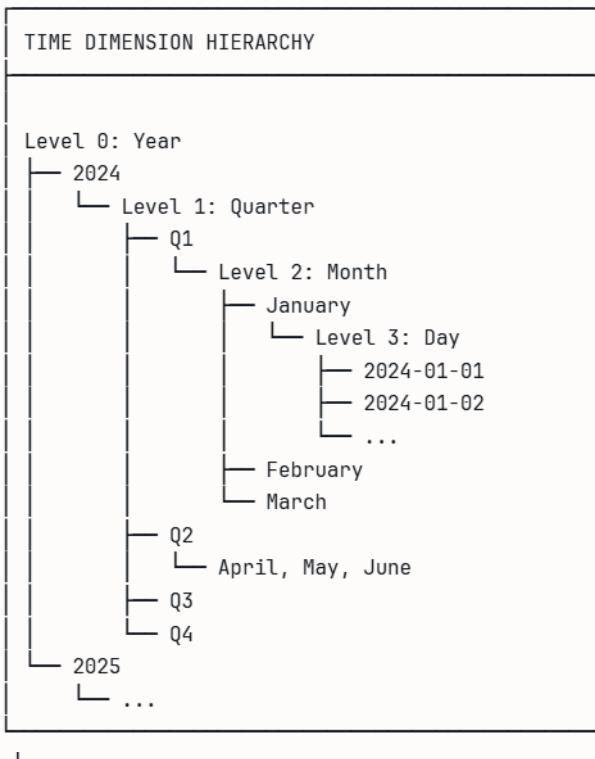
TOTAL COMPLETO: ~288 bytes

## 2.5.3.2.3.4.3.4: ¿QUÉ MÁS HAY EN MOLAP?

### Componentes Completos de un Cubo MOLAP

| COMPONENTES DE UN CUBO MOLAP                  |                                                                                                                                                                                                                                                                                                   |
|-----------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. METADATA (metadatos del cubo)              | <ul style="list-style-type: none"><li>— Nombre del cubo</li><li>— Dimensiones (nombre, tipo, cardinalidad)</li><li>— Medidas (nombre, tipo, aggregation func)</li><li>— Relaciones entre dimensiones</li><li>— Fechas de creación/actualización</li><li>— Configuración de agregaciones</li></ul> |
| 2. DIMENSION STORES (almacenes dimensionales) | <ul style="list-style-type: none"><li>— Todos los miembros de cada dimensión</li><li>— Todos los atributos de cada miembro</li><li>— Jerarquías (si existen)</li><li>— String dictionaries (compresión interna)</li></ul>                                                                         |
| 3. FACTS ARRAYS (arrays de hechos)            | <ul style="list-style-type: none"><li>— Un array por cada medida</li><li>— Granularidad más fina (detalle máximo)</li><li>— Valores numéricos solamente</li></ul>                                                                                                                                 |
| 4. AGGREGATIONS (agregaciones pre-calculadas) | <ul style="list-style-type: none"><li>— Totales por dimensión individual</li><li>— Totales por combinaciones de dimensiones</li><li>— Gran total (todas las dimensiones)</li><li>— Niveles de jerarquías (si existen)</li></ul>                                                                   |
| 5. INDEXES (índices de búsqueda)              | <ul style="list-style-type: none"><li>— Member ID → Offset mapping</li><li>— Attribute → Member IDs</li><li>— Natural key → Member ID</li><li>— B-trees para búsquedas rápidas</li></ul>                                                                                                          |
| 6. PARTITIONS (particiones - opcional)        | <ul style="list-style-type: none"><li>— División del cubo en pedazos</li><li>— Por rango de fechas</li><li>— Por región geográfica</li><li>— Para procesamiento paralelo</li></ul>                                                                                                                |
| 7. CALCULATIONS (cálculos - opcional)         | <ul style="list-style-type: none"><li>— Medidas calculadas (profit = sales-cost)</li><li>— Members calculados (YTD, QTD)</li><li>— Scripts MDX</li><li>— Fórmulas personalizadas</li></ul>                                                                                                        |
| 8. SECURITY (seguridad - opcional)            | <ul style="list-style-type: none"><li>— Roles y permisos</li><li>— Cell-level security</li><li>— Dimension-level security</li></ul>                                                                                                                                                               |
| 9. TRANSLATIONS (traducciones - opcional)     | <ul style="list-style-type: none"><li>— Nombres en múltiples idiomas</li><li>— Metadata localizada</li></ul>                                                                                                                                                                                      |
| 10. WRITEBACK (escritura - opcional)          | <ul style="list-style-type: none"><li>— Cambios de usuarios</li><li>— Simulaciones "what-if"</li></ul>                                                                                                                                                                                            |

## Jerarquías (Componente Importante)



Cada nivel tiene su propia agregación:

- Day level: facts originales (18 celdas)
- Month level: agregación (9 celdas)
- Quarter level: agregación (6 celdas)
- Year level: agregación (3 celdas)

## 2.5.3.2.3.4.3.5: 5. CÁLCULO DE OFFSET: ¿Qué Es Realmente?

### OFFSET = Posición en el Array 1D Lineal

#### DEFINICIÓN:

Offset es el ÍNDICE donde se encuentra un valor en el array 1D físico que representa el cubo ND.

Es la conversión de coordenadas N-dimensionales a una posición 1-dimensional.

`offset ∈ [0, total_cells - 1]`

donde `total_cells = dim1 × dim2 × ... × dimn`

### ¿Por Qué Existe el Offset?

#### PROBLEMA:

Lógicamente: Cubo 3D

`Array[product][customer][time]`

Físicamente: Memoria es 1D (lineal)

`[val0, val1, val2, val3, ..., valn]`

#### PREGUNTA:

¿Cómo mapear `[x][y][z] → posición en array?`

### RESPUESTA: OFFSET CALCULATION

#### SOLUCIÓN:

Coordenadas `[x][y][z]`

↓

Fórmula matemática

↓

`offset = número entero`

↓

`array[offset] = valor`

## La Fórmula de Offset: ¿Por Qué Es Así?

FÓRMULA GENERAL (N dimensiones):

$$\begin{aligned} \text{offset} = & x_0 \times (d_1 \times d_2 \times \dots \times d_{n-1}) + \\ & x_1 \times (d_2 \times d_3 \times \dots \times d_{n-1}) + \\ & x_2 \times (d_3 \times d_4 \times \dots \times d_{n-1}) + \\ & \dots \\ & x_{n-2} \times d_{n-1} + \\ & x_{n-1} \end{aligned}$$

Donde:

- $x_i$  = índice en dimensión i
- $d_i$  = tamaño de dimensión i

## ¿POR QUÉ esta fórmula?

Explicación Intuitiva:  
Exploración intuitiva

Imagina un edificio de apartamentos:

Dimensión 1: Piso (3 pisos)

Dimensión 2: Pasillo (2 pasillos por piso)

Dimensión 3: Apartamento (3 apts por pasillo)

Total:  $3 \times 2 \times 3 = 18$  apartamentos

Quiero ir a: Piso 1, Pasillo 1, Apto 2

↓      ↓      ↓  
[1]    [1]    [2]

¿Cuántos apartamentos he pasado?

PASO 1: Pisos completos antes del mío

- Estoy en piso 1
- Piso 0 completo = 1 piso  $\times$  6 apts/piso = 6 apts  
(6 apts/piso viene de 2 pasillos  $\times$  3 apts)

PASO 2: Pasillos completos en mi piso

- Estoy en pasillo 1
- Pasillo 0 completo = 1 pasillo  $\times$  3 apts = 3 apts

PASO 3: Apartamentos en mi pasillo

- Estoy en apartamento 2
- Apartamentos antes = 2

TOTAL:  $6 + 3 + 2 = 11$  apartamentos antes del mío

Offset = 11

array[11] = mi apartamento

### Fórmula Aplicada al Edificio:

```
offset = piso × (pasillos × apts) +
 pasillo × apts +
 apt
```



```
offset = 1 × (2 × 3) +
 1 × 3 +
 2
```

```
offset = 1 × 6 +
 1 × 3 +
 2
```

```
offset = 6 + 3 + 2 = 11
```

Correcto: array[11] es el apartamento [1][1][2]

## 2.5.3.2.3.4.3.6: 12 3.4 6. OFFSET PASO A PASO: 3D, 4D, 5D, 6D

### Ejemplo 1: Cubo 3D (nuestro ejemplo)

Dimensiones:

- Products: 3 (Laptop, Mouse, Keyboard)
- Customers: 2 (Alice, Bob)
- Time: 3 (Jan, Feb, Mar)

Total celdas:  $3 \times 2 \times 3 = 18$

Queremos: [1][1][1] (Mouse-Bob-Feb)

#### CÁLCULO PASO A PASO

$$\text{offset} = x \times (d_y \times d_z) + y \times d_z + z$$

Valores:

- x = 1 (Mouse - índice del producto)
- y = 1 (Bob - índice del cliente)
- z = 1 (Feb - índice del mes)
- d\_y = 2 (total clientes)
- d\_z = 3 (total meses)

Sustituyendo:

$$\text{offset} = 1 \times (2 \times 3) + 1 \times 3 + 1$$

$$\text{Paso 1: } (2 \times 3) = 6$$

$$\text{offset} = 1 \times 6 + 1 \times 3 + 1$$

$$\text{Paso 2: } 1 \times 6 = 6$$

$$\text{offset} = 6 + 1 \times 3 + 1$$

$$\text{Paso 3: } 1 \times 3 = 3$$

$$\text{offset} = 6 + 3 + 1$$

$$\text{Paso 4: } 6 + 3 + 1 = 10$$

$$\text{offset} = 10$$

array[10] contiene el valor de [1][1][1]

#### INTERPRETACIÓN:

- 6: Pasé 1 producto completo (Mouse es el 2do)

Cada producto tiene  $2 \times 3 = 6$  celdas

- 3: Pasé 1 cliente completo (Bob es el 2do)

Cada cliente tiene 3 celdas

- 1: Estoy en el 2do mes (Feb)

Pasé 1 mes antes

### Ejemplo 2: Cubo 4D

Dimensiones:  
- Products: 3  
- Customers: 2  
- Time: 3  
- Regions: 2 (North, South)

Total celdas:  $3 \times 2 \times 3 \times 2 = 36$

Queremos: [2][1][0][1] (Keyboard-Bob-Jan-South)

FÓRMULA 4D:  
offset =  $w \times (x\_size \times y\_size \times z\_size) +$   
 $x \times (y\_size \times z\_size) +$   
 $y \times z\_size +$   
z

Valores:

w = 2 (Keyboard - 3er producto, índice 2)  
x = 1 (Bob - 2do cliente, índice 1)  
y = 0 (Jan - 1er mes, índice 0)  
z = 1 (South - 2da región, índice 1)

Tamaños:

x\_size = 2 (clientes)  
y\_size = 3 (meses)  
z\_size = 2 (regiones)

Cálculo:

offset =  $2 \times (2 \times 3 \times 2) +$   
 $1 \times (3 \times 2) +$   
 $0 \times 2 +$   
1  
+

Paso 1:  $(2 \times 3 \times 2) = 12$   
offset =  $2 \times 12 + 1 \times (3 \times 2) + 0 \times 2 + 1$

Paso 2:  $2 \times 12 = 24$   
offset =  $24 + 1 \times 6 + 0 + 1$

Paso 3:  $1 \times 6 = 6$   
offset =  $24 + 6 + 0 + 1$

Paso 4:  $24 + 6 + 0 + 1 = 31$   
offset = 31

array[31] = Keyboard-Bob-Jan-South

### Ejemplo 3: Cubo 5D

Dimensiones:

- Products: 3
- Customers: 2
- Time: 3
- Regions: 2
- Channels: 2 (Online, Store)

Total celdas:  $3 \times 2 \times 3 \times 2 \times 2 = 72$

Queremos: [0][1][2][0][1] (Laptop-Bob-Mar-North-Store)

FÓRMULA 5D:  
offset =  $v \times (w\_size \times x\_size \times y\_size \times z\_size) +$   
 $w \times (x\_size \times y\_size \times z\_size) +$   
 $x \times (y\_size \times z\_size) +$   
 $y \times z\_size +$   
 $z$

Valores:

- $v = 0$  (Laptop)  
 $w = 1$  (Bob)  
 $x = 2$  (Mar)  
 $y = 0$  (North)  
 $z = 1$  (Store)

Tamaños:

- $w\_size = 2$   
 $x\_size = 3$   
 $y\_size = 2$   
 $z\_size = 2$

Cálculo detallado:

$$\begin{aligned} \text{Término 1: } & v \times (w\_size \times x\_size \times y\_size \times z\_size) \\ &= 0 \times (2 \times 3 \times 2 \times 2) \\ &= 0 \times 24 \\ &= 0 \end{aligned}$$

$$\begin{aligned} \text{Término 2: } & w \times (x\_size \times y\_size \times z\_size) \\ &= 1 \times (3 \times 2 \times 2) \\ &= 1 \times 12 \\ &= 12 \end{aligned}$$

$$\begin{aligned} \text{Término 3: } & x \times (y\_size \times z\_size) \\ &= 2 \times (2 \times 2) \\ &= 2 \times 4 \\ &= 8 \end{aligned}$$

$$\begin{aligned} \text{Término 4: } & y \times z\_size \\ &= 0 \times 2 \\ &= 0 \end{aligned}$$

Término 5: z

= 1

SUMA TOTAL:

offset = 0 + 12 + 8 + 0 + 1

offset = 21

array[21] = Laptop-Bob-Mar-North-Store



## Ejemplo 4: Cubo 6D

Dimensiones:

- Products: 3
- Customers: 2
- Time: 3
- Regions: 2
- Channels: 2
- Promotions: 3 (None, Discount, Bundle)

Total celdas:  $3 \times 2 \times 3 \times 2 \times 2 \times 3 = 216$

Queremos: [1][0][1][1][0][2]  
(Mouse-Alice-Feb-South-Online-Bundle)

FÓRMULA 6D:  
$$\begin{aligned} \text{offset} = & u \times (v_{\text{sz}} \times w_{\text{sz}} \times x_{\text{sz}} \times y_{\text{sz}} \times z_{\text{sz}}) + \\ & v \times (w_{\text{sz}} \times x_{\text{sz}} \times y_{\text{sz}} \times z_{\text{sz}}) + \\ & w \times (x_{\text{sz}} \times y_{\text{sz}} \times z_{\text{sz}}) + \\ & x \times (y_{\text{sz}} \times z_{\text{sz}}) + \\ & y \times z_{\text{sz}} + \\ & z \end{aligned}$$

Valores:

- $u = 1$  (Mouse)
- $v = 0$  (Alice)
- $w = 1$  (Feb)
- $x = 1$  (South)
- $y = 0$  (Online)
- $z = 2$  (Bundle)

Tamaños:

$v_{\text{sz}} = 2, w_{\text{sz}} = 3, x_{\text{sz}} = 2, y_{\text{sz}} = 2, z_{\text{sz}} = 3$

Cálculo:

$$\begin{aligned} \text{Término 1: } & u \times (2 \times 3 \times 2 \times 2 \times 3) \\ & = 1 \times 72 \\ & = 72 \end{aligned}$$

$$\begin{aligned} \text{Término 2: } & v \times (3 \times 2 \times 2 \times 3) \\ & = 0 \times 36 \\ & = 0 \end{aligned}$$

$$\begin{aligned} \text{Término 3: } & w \times (2 \times 2 \times 3) \\ & = 1 \times 12 \\ & = 12 \end{aligned}$$

$$\begin{aligned} \text{Término 4: } & x \times (2 \times 3) \\ & = 1 \times 6 \\ & = 6 \end{aligned}$$

Término 5:  $y \times 3$

=  $0 \times 3$

= 0

Término 6: z

= 2

SUMA:

offset = 72 + 0 + 12 + 6 + 0 + 2

offset = 92

array[92] = Mouse-Alice-Feb-South-Online-Bundle

## 2.5.3.2.3.4.3.7: Tabla de Patrones: Offset por Dimensiones + Resumen Final: Lo Que Realmente Es el Offset

| Dims | Patrón de la Fórmula                                                                                                                              |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| 1D   | $offset = x$                                                                                                                                      |
| 2D   | $offset = x \times d_y + y$                                                                                                                       |
| 3D   | $offset = x \times (d_y \times d_z) + y \times d_z + z$                                                                                           |
| 4D   | $offset = w \times (d_x \times d_y \times d_z) + x \times (d_y \times d_z) + y \times d_z + z$                                                    |
| 5D   | $offset = v \times (d_w \times d_x \times d_y \times d_z) + w \times (d_x \times d_y \times d_z) + x \times (d_y \times d_z) + y \times d_z + z$  |
| ND   | $offset = \sum (x_i \times \prod (d_j \text{ for } j > i))$<br>(suma de cada coordenada multiplicada por el producto de dimensiones subsecuentes) |

## ⌚ RESUMEN FINAL: Lo Que Realmente Es el Offset

| OFFSET ES:                                                                                                                                                                                                                                                      |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. Un ÍNDICE en el array 1D físico<br>2. Una TRADUCCIÓN de coordenadas ND → 1D<br>3. Una POSICIÓN ÚNICA para cada celda<br>4. Un NÚMERO entre 0 y (total_cells - 1)<br>5. Un CÁLCULO matemático determinístico:<br>Mismas coordenadas → Mismo offset<br>SIEMPRE |

La fórmula existe porque:

- ✓ La memoria es lineal (1D)
- ✓ Los cubos son N-dimensionales
- ✓ Necesitamos un mapeo ÚNICO y EFICIENTE
- ✓ La fórmula garantiza NO colisiones
- ✓ Es calculable en O(N) donde N = núm dimensiones

## 2.5.3.2.3.4.4: ⚡ MOLAP: ¿Siempre desde ROLAP? - Aclaración Crítica

⚠ CORRECCIÓN IMPORTANTE: NO Siempre ROLAP → MOLAP

Excelente pregunta. Necesito aclarar esto porque NO es obligatorio pasar de ROLAP a MOLAP.

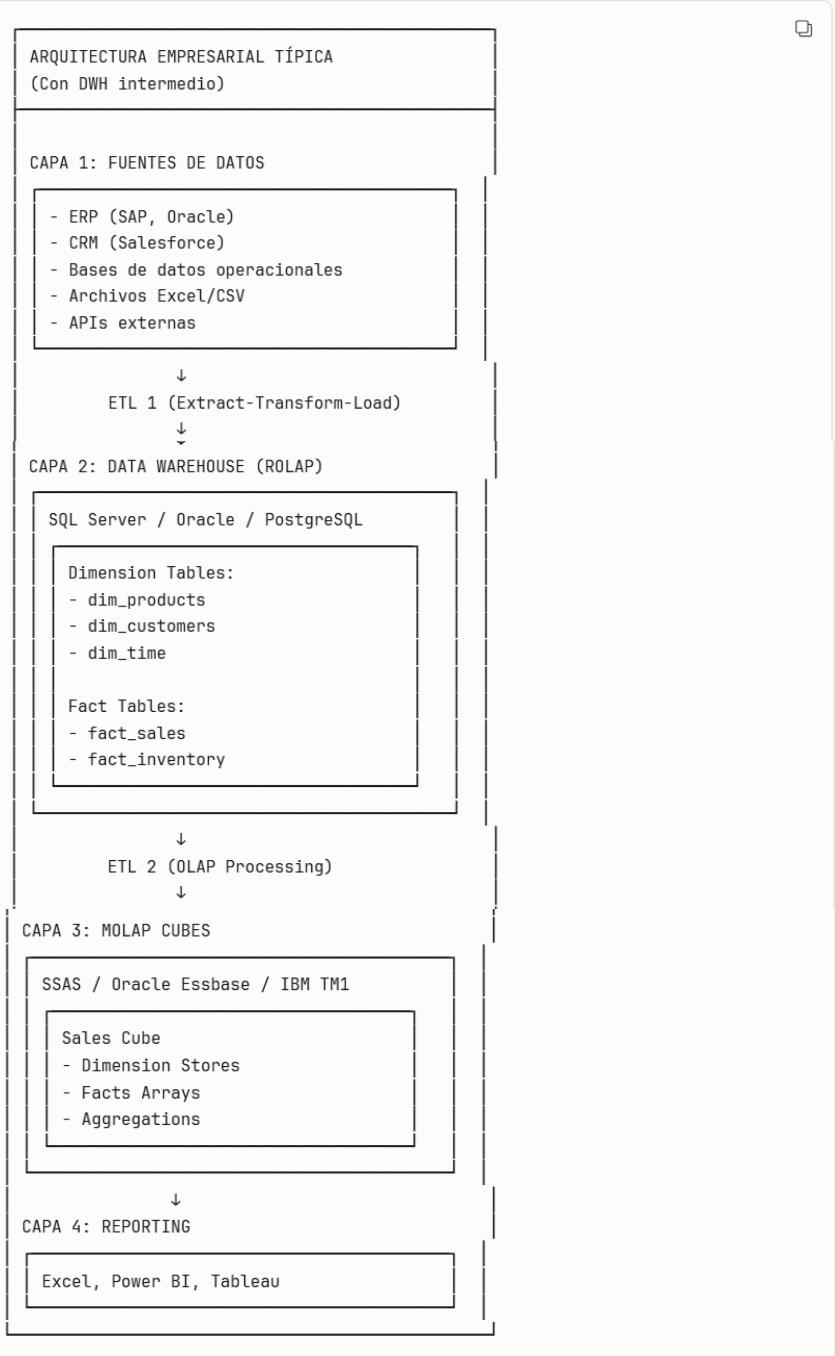
### 2.5.3.2.3.4.4.1: DIMENSION STORES: Múltiples Orígenes

Dimension Stores pueden venir de CUALQUIER fuente de datos:

| FUENTES DE DATOS PARA MOLAP                                                  |                                                                           |
|------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| <input checked="" type="checkbox"/> OPCIÓN 1: Data Warehouse (ROLAP)         | DWH → Dimension Tables → MOLAP Cube<br>(El más común en empresas grandes) |
| <input checked="" type="checkbox"/> OPCIÓN 2: Archivos CSV/Excel             | CSV Files → ETL → MOLAP Cube<br>(Común en empresas pequeñas/medianas)     |
| <input checked="" type="checkbox"/> OPCIÓN 3: Bases de Datos Transaccionales | OLTP DB → ETL directo → MOLAP Cube<br>(Sin DWH intermedio)                |
| <input checked="" type="checkbox"/> OPCIÓN 4: APIs / Web Services            | API → JSON/XML → ETL → MOLAP Cube<br>(Datos de servicios externos)        |
| <input checked="" type="checkbox"/> OPCIÓN 5: Data Lakes                     | Parquet/ORC → ETL → MOLAP Cube<br>(Big Data sources)                      |
| <input checked="" type="checkbox"/> OPCIÓN 6: Archivos de texto / Logs       | Log Files → Parser → MOLAP Cube<br>(Análisis de logs)                     |
| <input checked="" type="checkbox"/> OPCIÓN 7: Otras fuentes                  | - Google Sheets<br>- Salesforce<br>- SAP<br>- Cualquier fuente tabular    |

## 2.5.3.2.3.4.4.2: Arquitecturas Reales: Con y Sin ROLAP

### ARQUITECTURA 1: Con Data Warehouse (COMÚN)



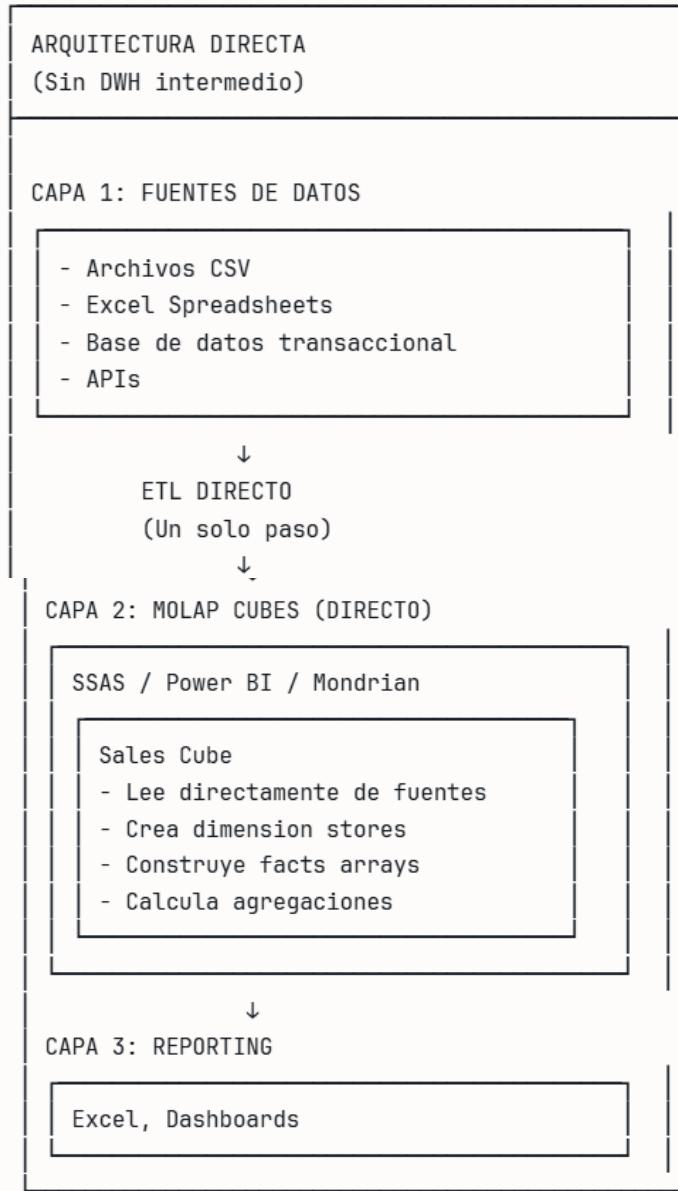
#### VENTAJAS:

- ✓ DWH sirve múltiples propósitos (no solo MOLAP)
- ✓ Queries SQL disponibles también
- ✓ Data warehouse como "single source of truth"
- ✓ Separación de responsabilidades

#### DESVENTAJAS:

- ✗ Más complejo
- ✗ Más costoso (2 sistemas)
- ✗ Más lento (2 ETLs)

### ARQUITECTURA 2: Sin Data Warehouse (MAS SIMPLE)



#### VENTAJAS:

- Más simple
- Más barato
- Más rápido de implementar
- Suficiente para empresas pequeñas/medianas

#### DESVENTAJAS:

- Solo tienes MOLAP (no SQL queries ad-hoc)
- Menos flexible
- Toda la lógica en el cubo

#### 2.5.3.2.3.4.4.3: EJEMPLOS REALES: ¿Es Factible MOLAP?

¡SÍ! MOLAP es MUY usado en el mundo real

## EMPRESAS QUE USAN MOLAP

### 1. Microsoft

Producto: SQL Server Analysis Services (SSAS)  
Usado por: Miles de empresas Fortune 500

### 2. Oracle

Producto: Essbase  
Usado por: P&G, Coca-Cola, Walmart

### 3. IBM

Producto: Cognos TM1  
Usado por: Shell, HP, AstraZeneca

### 4. SAP

Producto: SAP BW (incluye MOLAP)  
Usado por: BMW, Nestlé, Unilever

### 5. Microsoft

Producto: Power BI (backend MOLAP)  
Usado por: 97% Fortune 500

## Casos de Uso Reales:

### CASO 1: Retailer Grande (Walmart-style)

Arquitectura:

Fuentes → DWH (ROLAP) → MOLAP Cubes

Por qué MOLAP:

- 100,000+ productos
- 5,000+ tiendas
- 365 días × 5 años = 1,825 días
- Total: 912,500,000 combinaciones

Queries típicos:

"Ventas por región en último trimestre"

ROLAP: 30 segundos

MOLAP: 0.1 segundos

MOLAP es CRÍTICO para performance

### CASO 2: Empresa Mediana (Sin DWH)

Arquitectura:

CSV Files → MOLAP Cube directo

Datos:

- Sales.csv
- Products.csv
- Customers.csv

Proceso:

1. Leer CSVs
2. Crear dimension stores
3. Crear facts array
4. Calcular agregaciones

NO necesita DWH

Directamente de archivos a MOLAP

### CASO 3: Startup (Power BI)

Arquitectura:

PostgreSQL → Power BI (MOLAP interno)

Proceso:

1. Power BI conecta a PostgreSQL
2. Power BI construye modelo (MOLAP interno)
3. Usuarios crean reportes

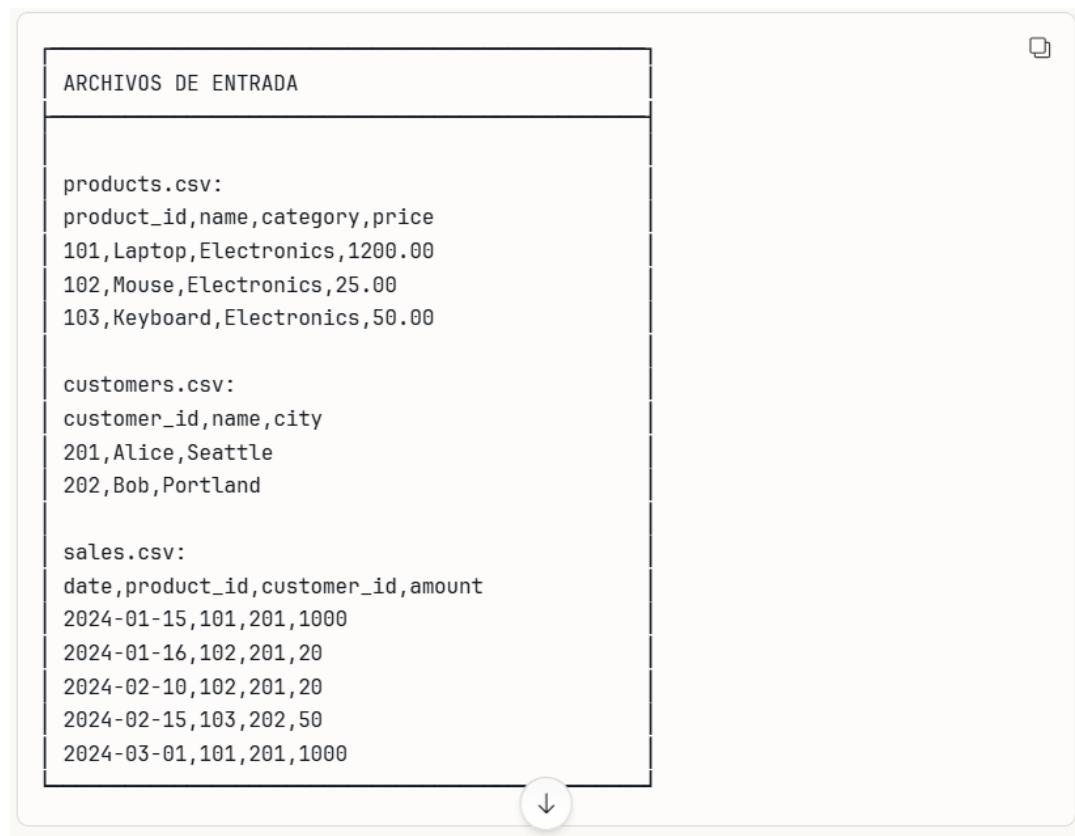
NO hay DWH separado

NO hay SSAS separado

MOLAP "escondido" dentro de Power BI

## 2.5.3.2.3.4.4.4: Ejemplo Práctico: CSV Directo a MOLAP (con código)

Escenario: Empresa pequeña con solo archivos CSV



## Proceso: CSV → MOLAP (Sin ROLAP)

```
Código conceptual (Python + custom MOLAP engine)

import csv
import numpy as np

PASO 1: Leer archivos CSV
products = read_csv('products.csv')
customers = read_csv('customers.csv')
sales = read_csv('sales.csv')

PASO 2: Crear Dimension Stores (directo de CSV)
products_store = DimensionStore('products')
for row in products:
 products_store.add_member({
 'product_key': row['product_id'], # 101, 102, 103
 'name': row['name'], # Laptop, Mouse...
 'category': row['category'], # Electronics
 'price': float(row['price']) # 1200.00, 25.00...
 })
Resultado: Members 0, 1, 2 creados

customers_store = DimensionStore('customers')
for row in customers:
 customers_store.add_member({
 'customer_key': row['customer_id'], # 201, 202
 'name': row['name'], # Alice, Bob
 'city': row['city'] # Seattle, Portland
 })
Resultado: Members 0, 1 creados

PASO 3: Crear Time Dimension (derivada de sales)
time_store = DimensionStore('time')
unique_dates = set(row['date'] for row in sales)
for date in sorted(unique_dates):
 time_store.add_member({
 'date': date,
 'year': extract_year(date),
 'month': extract_month(date),
 'day': extract_day(date)
 })

PASO 4: Crear Facts Array
Dimensiones: 3 products × 2 customers × N dates
cube_shape = (
 products_store.size(), # 3
 customers_store.size(), # 2
 time_store.size() # depende de fechas únicas
)

facts_array = np.zeros(cube_shape, dtype=np.float64)

PASO 5: Poblar Facts Array (leer desde sales.csv)
for row in sales:
 # Traducir keys → member IDs
 product_id = products_store.get_member_id(row['product_id'])
 customer_id = customers_store.get_member_id(row['customer_id'])
 time_id = time_store.get_member_id(row['date'])

 # Guardar en array
 facts_array[product_id][customer_id][time_id] = row['amount']

PASO 6: Calcular Agregaciones
aggregations = calculate_aggregations(facts_array)

PASO 7: Guardar Cubo MOLAP
save_cube({
 'dimensions': [products_store, customers_store, time_store],
 'facts': facts_array,
 'aggregations': aggregations
}, 'sales_cube.mdb')

print("✅ Cubo MOLAP creado desde CSV (sin ROLAP)")
```

## Resultado: Estructura MOLAP creada\*

```
sales_cube.mdb/
└── dimensions/
 ├── products.dim ← Desde products.csv
 ├── customers.dim ← Desde customers.csv
 └── time.dim ← Derivado de sales.csv

└── facts/
 └── sales_amount.dat ← Desde sales.csv

└── aggregations/
 ├── by_product.agg
 ├── by_customer.agg
 └── grand_total.agg
```

- ✗ NO hubo DWH intermedio
- ✗ NO hubo dimension tables relacionales
- ✓ Directamente CSV → MOLAP

...

## 2.5.3.2.3.4.4.5: ¿Cuándo Usar ROLAP como Paso Intermedio?

### USAR DWH (ROLAP) INTERMEDIO CUANDO:

- Empresa grande (Fortune 500)
  - Múltiples fuentes de datos
  - Múltiples departamentos
  - Necesitas SQL queries también
- Datos complejos
  - Transformaciones complejas
  - Limpieza de datos extensa
  - Consolidación de múltiples fuentes
- Multiple uses
  - No solo OLAP
  - También Machine Learning
  - También reportes ad-hoc
  - También auditoría
- Regulaciones
  - Auditoría requerida
  - Trazabilidad completa
  - Data lineage



### IR DIRECTO A MOLAP CUANDO:

- Empresa pequeña/medianas
  - Pocas fuentes de datos
  - Datos ya limpios
  - Budget limitado
- Datos simples
  - CSV/Excel bien estructurados
  - Sin transformaciones complejas
  - Una sola fuente o pocas fuentes
- Solo necesitas OLAP
  - No necesitas SQL ad-hoc
  - Solo dashboards/reportes
  - Análisis multidimensional únicamente
- Prototipo/POC rápido
  - Necesitas resultados rápido
  - Demostración de concepto
  - No es producción aún



## 2.5.3.2.3.4.4.6:💡 Tecnologías Reales que Permiten Ambos

### Microsoft SQL Server Analysis Services (SSAS)

#### SSAS: Flexible en Fuentes de Datos

Puede leer de:

- SQL Server (DWH)
- Oracle Database
- Excel Files
- CSV Files
- Azure SQL
- APIs (via custom)
- Cualquier ODBC/OLEDB

Arquitectura típica:

DWH → SSAS (común)

Arquitectura alternativa:

CSV → SSAS (también válido)

Excel → SSAS (también válido)

OLTP DB → SSAS directo (también válido)

### Power BI (MOLAP "escondido")

#### POWER BI: MOLAP Transparente



Lo que el usuario ve:

"Conecto a PostgreSQL y creo reportes"

Lo que realmente pasa internamente:

1. Power BI lee PostgreSQL
2. Power BI construye modelo dimensional
3. Power BI crea MOLAP interno (VertiPaq)
4. Power BI guarda en .pbix (MOLAP comprimido)

Fuentes soportadas:

- 150+ conectores
- SQL Server, PostgreSQL, MySQL
- Excel, CSV
- SharePoint
- Google Analytics
- Salesforce, Dynamics
- Web APIs

✗ NO necesitas DWH

MOLAP automático en background

## 2.5.3.2.3.4.4.7: CONCLUSIÓN: Respondiendo Tu Pregunta + Resumen Final

### 1. "¿Dimension Stores siempre de ROLAP?"

NO, NO siempre de ROLAP

Dimension Stores pueden venir de:

- DWH (ROLAP) ← común en empresas grandes
- CSV/Excel ← común en empresas pequeñas
- OLTP directo ← común en startups
- APIs ← común en SaaS
- Cualquier fuente tabular

...

### 2. "¿Cómo llegamos a ROLAP → MOLAP?"

#### Históricamente (1990s-2000s):

1. Empresas tenían DWH (ROLAP)
2. ROLAP era lento para analytics
3. Inventaron MOLAP como capa de optimización
4. Arquitectura típica: Fuentes → DWH → MOLAP

#### Modernamente (2010s-presente):

1. Muchas empresas NO tienen DWH
2. Van directo de fuentes a MOLAP
3. Power BI, Tableau hacen MOLAP transparente
4. Arquitectura: Fuentes → MOLAP directo

### 3. "¿Es realmente factible?"

SÍ, MUY factible

#### Evidencia:

- Microsoft: 250,000+ clientes SSAS
- Power BI: 15 millones de usuarios
- Oracle Essbase: 18,000+ empresas
- SAP BW: 40,000+ instalaciones

MOLAP es una tecnología MADURA y PROBADA  
usado por MILES de empresas Fortune 500

### 4. "¿No solo se da MOLAP sino que tiene que pasar de ROLAP?"

 NO tiene que pasar de ROLAP

 Hay 2 caminos principales:

CAMINO 1 (Empresas grandes):

Fuentes → DWH (ROLAP) → MOLAP

Razón: Necesitan DWH para múltiples usos

CAMINO 2 (Empresas pequeñas/medianas):

Fuentes → MOLAP directo

Razón: Solo necesitan analytics, no DWH completo

AMBOS son válidos

AMBOS son comunes

AMBOS funcionan bien

---

### Resumen Final

DIMENSION STORES vienen de CUALQUIER fuente:

- DWH (ROLAP)  Común
- CSV/Excel  Común
- OLTP DB  Común
- APIs  Común
- Otros  Común

MOLAP es factible y usado ampliamente:

- Fortune 500 
- Empresas medianas 
- Startups 
- Gobierno 

NO necesitas ROLAP primero:

- Si tienes DWH → Úsallo 
- Si no tienes DWH → Ve directo 
- Ambos caminos funcionan 

**2.5.3.2.5: ⚡ ROLAP vs MOLAP vs HOLAP: Consultas, Estructuras y Convergencia**

**2.5.3.2.5.1: 📊 CONSULTAS AD-HOC: ROLAP vs MOLAP**

**¿Qué es una consulta Ad-Hoc?**

### AD-HOC QUERY (Consulta improvisada)

#### Definición:

Una consulta que NO fue prevista/planeada durante el diseño del sistema

#### Características:

- Escrita en el momento (no pre-definida)
- Estructura impredecible
- Puede usar cualquier combinación de campos
- No necesariamente usa dimensiones del cubo
- Puede requerir JOINs complejos

## Ejemplo Práctico: Ad-Hoc en ROLAP



```
-- CONSULTA AD-HOC #1: Análisis no previsto
-- "Clientes que compraron laptops Dell en Seattle
-- pero nunca compraron accesorios"

SELECT DISTINCT
 c.customer_id,
 c.first_name,
 c.last_name,
 c.email,
 COUNT(DISTINCT f1.transaction_id) as laptop_purchases,
 SUM(f1.amount) as total_spent_laptops
FROM
 dim_customers c
 INNER JOIN fact_sales f1
 ON c.customer_id = f1.customer_id
 INNER JOIN dim_products p1
 ON f1.product_id = p1.product_id
 INNER JOIN dim_geography g
 ON c.city_id = g.city_id
WHERE
 -- Filtros complejos no dimensionales
 p1.brand = 'Dell'
 AND p1.subcategory = 'Laptops'
 AND g.city = 'Seattle'
 AND c.customer_id NOT IN (
 -- Subquery: Nunca compraron accesorios
 SELECT DISTINCT customer_id
 FROM fact_sales f2
 INNER JOIN dim_products p2
 ON f2.product_id = p2.product_id
 WHERE p2.category = 'Accessories'
)
 -- Filtro por fecha NO en dimensión tiempo
 AND f1.transaction_date BETWEEN
 DATEADD(month, -3, GETDATE()) AND GETDATE()
 -- Filtro por hora (NO está en cubo típico)
 AND DATEPART(hour, f1.transaction_timestamp) BETWEEN 18 AND 22
GROUP BY
 c.customer_id,
 c.first_name,
 c.last_name,
 c.email
HAVING
 -- Condición compleja post-agregación
 SUM(f1.amount) > (
 SELECT AVG(total)
 FROM (
 SELECT SUM(amount) as total
 FROM fact_sales
 GROUP BY customer_id
) avg_spending
)
ORDER BY
 total_spent_laptops DESC;
```

## -- **ROLAP: Ejecuta perfectamente**

- - JOINs arbitrarios permitidos
- - Subqueries anidadas OK
- - Filtros por cualquier campo
- - Agregaciones complejas
- Tiempo: 5-30 segundos (dependiendo de índices)

## -- **MOLAP: IMPOSIBLE**

- - No puede hacer este tipo de query
- - No tiene "transaction\_timestamp" con hora
- - No puede hacer NOT IN con subquery compleja
- - No tiene concepto de JOINs tradicionales

---

## **El Mismo Análisis en MOLAP**



### **SELECT**

```
[Measures].[Sales Amount] ON COLUMNS,
[Customers].[Customer].[Customer] ON ROWS
FROM [Sales Cube]
WHERE (
 [Products].[Brand].[Dell],
 [Products].[SubCategory].[Laptops],
 [Geography].[City].[Seattle]
)
```

### **PROBLEMAS:**

- No puede filtrar por hora del día (no está en cubo)
- No puede hacer "NOT IN" con subconsulta compleja
- No puede comparar con promedio de otros clientes
- No puede filtrar "últimos 3 meses desde hoy"  
(Time dimension tiene fechas fijas, no relativas)

### **SOLUCIÓN EN MOLAP:**

Tienes que:

1. Exportar datos a SQL
  2. Hacer análisis en ROLAP
  3. O rediseñar el cubo para incluir estas dimensiones  
(pero eso multiplica el tamaño exponencialmente)
-

## Consultas Comunes: ROLAP vs MOLAP

### TIPO 1: Consultas Dimensionales Simples

Query: "Ventas por producto y mes"

ROLAP:

```
SELECT
 p.product_name,
 t.month_name,
 SUM(f.amount) AS total
FROM fact_sales f
JOIN dim_products p ON f.product_id = p.id
JOIN dim_time t ON f.date_id = t.id
GROUP BY p.product_name, t.month_name
```

Tiempo: 10-30 segundos

Razón: Escanea millones de filas + JOINs

MOLAP:

```
SELECT [Measures].[Sales] ON 0,
 [Products].[Product] * [Time].[Month] ON 1
FROM [Sales]
```

Tiempo: 0.01-0.1 segundos

Razón: Lee agregación pre-calculada

GANADOR: MOLAP (1000x más rápido)

### TIPO 2: Consultas con Filtros Complejos

Query: "Productos donde nombre contiene 'wireless'  
Y precio entre \$20-\$50  
Y launch\_date en últimos 6 meses  
Y supplier está en lista específica"

ROLAP:

```
SELECT p.product_name, SUM(f.amount)
FROM fact_sales f
JOIN dim_products p ON f.product_id = p.id
WHERE p.name LIKE '%wireless%'
 AND p.price BETWEEN 20 AND 50
 AND p.launch_date > DATEADD(month, -6, GETDATE())
 AND p.supplier_id IN (101, 105, 234, 567)
GROUP BY p.product_name
```

Tiempo: 5-15 segundos

Razón: Índices en múltiples campos ayudan

MOLAP:

DIFÍCIL/IMPOSIBLE si estos atributos no están en el cubo como dimensiones separadas

Si están en cubo:

- Filtrar manualmente por cada valor
- No hay operador LIKE '%wireless%'
- No hay DATEADD(month, -6, GETDATE())

GANADOR: ROLAP (mayor flexibilidad)

### TIPO 3: Análisis de Cohortes (Cohortes)

Query: "Retención de clientes por mes de registro"

ROLAP:

```
WITH cohorts AS (
 SELECT
 customer_id,
 DATE_TRUNC('month', first_purchase) AS cohort
 FROM (
 SELECT
 customer_id,
 MIN(purchase_date) AS first_purchase
 FROM fact_sales
 GROUP BY customer_id
)
),
monthly_activity AS (
 SELECT
 c.cohort,
 DATE_TRUNC('month', f.purchase_date) AS month,
 COUNT(DISTINCT f.customer_id) AS active_users
 FROM cohorts c
 JOIN fact_sales f ON c.customer_id = f.customer_id
 GROUP BY c.cohort, month
)
SELECT
 cohort,
 month,
 active_users,
 active_users * 100.0 / first_month_users AS retention
FROM monthly_activity
```

Tiempo: 30-60 segundos

Razón: CTEs complejos, window functions

MOLAP:

IMPOSIBLE hacer en MDX estándar

Necesitarías pre-calcular cohorts en ETL

GANADOR: ROLAP (única opción)

#### TIPO 4: Drill-Through a Detalle



Query: "Ver todas las transacciones individuales que componen las ventas de Laptop-Dell-Q1"

ROLAP:

```
SELECT *
FROM fact_sales f
JOIN dim_products p ON f.product_id = p.id
JOIN dim_time t ON f.date_id = t.id
WHERE p.product_name = 'Dell Laptop'
AND t.quarter = 'Q1'
AND t.year = 2024
ORDER BY f.transaction_date
```

Tiempo: 2-10 segundos

Resultado: Filas individuales (transacciones)

MOLAP:

PUEDE hacer drill-through PERO:

- Tiene que ir a ROLAP backend
- Más lento porque tiene 2 pasos:
  1. MOLAP identifica qué filas
  2. ROLAP las trae

Tiempo: 5-20 segundos

GANADOR: ROLAP (acceso directo)

**TIPO 5:** Dashboard Estándar (Métricas del Negocio)

Query: "Ventas del día, semana, mes, año  
Top 10 productos  
Ventas por región  
Tendencia Últimos 30 días"

ROLAP:

4 queries separadas:

- Query 1: Agregación diaria/semanal/mensual/anual
- Query 2: TOP 10 con ORDER BY
- Query 3: GROUP BY región
- Query 4: Últimos 30 días con DATE filters

Tiempo TOTAL: 20-60 segundos (suma de 4 queries)

MOLAP:

4 queries MDX simples:

- Query 1: Lee agregaciones pre-calculadas
- Query 2: TopCount([Products], 10, [Sales])
- Query 3: Lee agregación by región
- Query 4: Últimos 30 members de time

Tiempo TOTAL: 0.1-0.5 segundos

GANADOR: MOLAP (100x más rápido)

## 2.5.3.2.5.2: HOLAP: El Híbrido Completo + HOLAP: Estructura Completa, Row vs Column Storage, y Convergencia

### ¿Qué es HOLAP?

HOLAP = Hybrid OLAP

#### Definición:

Sistema que combina ROLAP y MOLAP en EL MISMO CUBO

#### Idea central:

- Agregaciones → MOLAP (rápido)
- Datos detallados → ROLAP (flexible)

#### Objetivo:

- Velocidad de MOLAP para reportes
- Flexibilidad de ROLAP para drill-down
- Menor uso de disco que MOLAP puro

## 2.5.3.2.5.2.1: HOLAP: Arquitectura Completa

### Estructura Física de un Cubo HOLAP

```
/holap_cube/
└── /metadata/
 ├── cube_definition.xml
 | Define:
 | - Dimensiones (iguales que MOLAP)
 | - Medidas
 | - Storage mode POR PARTICIÓN
 | - Aggregation design
```

```
 └── partition_storage_map.xml


```
            Partition: 2024_Current  
                └── Storage: ROLAP  
                └── Date range: 2024-01-01 to NOW  
                └── Reason: Datos cambian diario  
                └── Source: DWH.fact_sales_2024  
  
            Partition: 2023_Archive  
                └── Storage: MOLAP  
                └── Date range: 2023-01-01 to ...  
                └── Reason: Datos estáticos  
                └── Source: Local .dat files  
  
            Partition: 2022_Archive  
                └── Storage: MOLAP  
                └── ...
```


```

```
└── aggregation_storage_map.xml
 Decide qué agregaciones en MOLAP vs ROLAP

└── /dimensions/ # SIEMPRE MOLAP
 ├── products.dim # (igual que MOLAP puro)
 ├── customers.dim
 ├── geography.dim
 └── time.dim
```

Nota: Dimensions SIEMPRE se guardan localmente  
en formato MOLAP para velocidad

```
└── /aggregations/ # MOLAP Storage
 └── /precomputed/ # Agregaciones pre-calculadas
 ├── by_product.agg # MOLAP format
 ├── by_customer.agg
 ├── by_geography.agg
 ├── by_time_month.agg # Por mes
 ├── by_time_quarter.agg # Por trimestre
 ├── by_time_year.agg # Por año
 ├── by_product_time.agg
 └── grand_total.agg

 └── aggregation_index.idx
 Mapa de qué agregaciones existen
```

```
└── /facts/
 └── /molap_partitions/ # Facts en MOLAP
 ├── partition_2022.dat # Año completo 2022
 | Storage: MOLAP
 | Size: 15 GB (comprimido)
 | Reason: Datos históricos estáticos
 | Last updated: 2023-01-01
 | Access pattern: Read-only
 |
 ├── partition_2023.dat # Año completo 2023
 | Storage: MOLAP
 | Size: 18 GB
 | Reason: Datos históricos
 |
 └── partition_metadata.xml
 Índices y metadata
```

```
└── /rolap_connections/ # Facts en ROLAP
 └── connection_2024_current.xml
 Connection String:
 Server=dwh.company.com
 Database=sales_dwh
 Table=fact_sales
 Filter=year=2024 AND
 month>=1

 Storage Mode: ROLAP
 Real-time: YES
 Cache TTL: 5 minutes

 Query delegation:
 - Aggregations: Push to SQL
 - Filters: Push to SQL
 - GROUP BY: Push to SQL

 └── connection_staging.xml
 Para datos en staging (pre-producción)
```

```
└── /cache/
 └── rolap_query_cache/ # Cache de queries ROLAP
 ├── query_abc123.cache
 └── query_def456.cache

 └── molap_facts_cache/ # Cache de facts MOLAP
 └── chunks_*.cache

└── /indexes/
 ├── product_indexes.idx # B-trees
 ├── customer_indexes.idx
 └── time_indexes.idx
```

### **2.5.3.2.5.2.2: Storage Decisioning Engine: Motor de toma de decisiones de almacenamiento**

#### **Cómo HOLAP Decide Dónde Guardar Datos**

##### **STORAGE DECISION ALGORITHM: Algoritmo de Decisión de Almacenamiento**

Para cada partición del cubo, evaluar:

###### **CRITERIO 1: Frecuencia de cambio**

¿Los datos cambian frecuentemente?

SÍ (cambios diarios/horarios)

↳ ROLAP

Razón: No procesar cubo cada hora

NO (datos históricos, inmutables)

↳ MOLAP

Razón: Optimizar velocidad

## **CRITERIO 2: Tamaño de datos**

¿Cuántos GB de datos detallados?

< 50 GB

↳ MOLAP OK

Razón: Cabe en disco razonablemente

50-500 GB

↳ HOLAP (aggs MOLAP, facts ROLAP)

Razón: Balance espacio/velocidad

> 500 GB

↳ ROLAP

Razón: MOLAP sería demasiado grande

## **CRITERIO 3: Patrón de acceso**

¿Cómo se accede a estos datos?

Queries agregadas (90% de queries)

↳ MOLAP con agregaciones

Razón: Pre-cálculo vale la pena

Queries detalladas (drill-through)

↳ ROLAP

Razón: Necesita flexibilidad SQL

Mix (agregadas + detalladas)

↳ HOLAP

Razón: Mejor de ambos mundos

## **CRITERIO 4: Requisitos de latencia**

¿Qué tan rápido necesita ser?

< 100ms (dashboard en vivo)

↳ MOLAP obligatorio

< 5s (reportes interactivos)

↳ MOLAP preferible, HOLAP aceptable

> 5s (análisis profundo)

↳ ROLAP aceptable

#### **CRITERIO 5: Necesidad de drill-through**

¿Los usuarios necesitan ver detalles?

SÍ, frecuentemente

↳ ROLAP o HOLAP

Razón: MOLAP malo para detalles

NO, solo agregados

↳ MOLAP puro

#### **2.5.3.2.5.2.2 Ejemplo de Decisión Real**

**ESCENARIO:** Cubo de Ventas con 5 años de historia

## Análisis por año:

### AÑO 2020 (hace 4 años)

Datos: 365 días × 1000 products × 50K customers  
Tamaño: 18.25M celdas × 8 bytes = 146 MB  
Cambios: NINGUNO (datos cerrados)  
Acceso: 5% de queries (poco usado)  
Drill-through: Raro

#### DECISIÓN: MOLAP

- Facts: MOLAP (.dat comprimido)
- Agregaciones: Todas pre-calculadas
- Cache priority: LOW

### AÑO 2021-2023 (histórico reciente)

Datos: 3 años × similares  
Tamaño: ~450 MB  
Cambios: Ocasionales (correcciones)  
Acceso: 30% de queries (comparaciones YoY)  
Drill-through: Moderado

#### DECISIÓN: MOLAP

- Facts: MOLAP con compresión alta
- Agregaciones: Pre-calculadas
- Cache priority: MEDIUM
- Update: Re-process si hay correcciones

### AÑO 2024 - Q1, Q2, Q3 (histórico completado)

Datos: 9 meses × similares  
Tamaño: ~135 MB  
Cambios: Ninguno (trimestres cerrados)  
Acceso: 40% de queries (análisis reciente)  
Drill-through: Frecuente

#### DECISIÓN: MOLAP

- Facts: MOLAP
- Agregaciones: Todas
- Cache priority: HIGH

### AÑO 2024 - Q4 (mes actual + próximos)

Datos: ~90 días en progreso  
Tamaño: ~45 MB (detalle)  
Cambios: DIARIOS (ventas nuevas cada hora)  
Acceso: 60% de queries (foco principal)  
Drill-through: MUY frecuente  
Latencia requerida: < 1 segundo

#### DECISIÓN: HOLAP

- Facts detallados: ROLAP
  - Query directo a DWH (real-time)
- Agregaciones: MOLAP
  - Re-calcular cada noche
- Cache:
  - ROLAP query cache: 5 min TTL
  - MOLAP aggs cache: Always in RAM

#### Ventajas:

- Dashboards usan aggs MOLAP (0.1s)
- Drill-through usa ROLAP (2s, aceptable)
- Datos siempre actualizados
- No reprocesar cubo cada hora

## RESUMEN DE STORAGE:

### Total facts detallados:

- MOLAP: 731 MB (años 2020-2024 Q1-Q3)

- ROLAP: 45 MB (2024 Q4 actual)

- Ratio: 94% MOLAP, 6% ROLAP

#### **Total agregaciones:**

- MOLAP: 200 MB (todas pre-calculadas)

#### **Ventajas vs MOLAP puro:**

- Tamaño en disco: 931 MB vs 1200 MB (ahorro 22%)
- Velocidad queries agregadas: Igual (usa MOLAP aggs)
- Actualización: Sin reprocesar (ROLAP real-time)
- Drill-through: Más rápido (SQL directo vs MOLAP)

#### **Ventajas vs ROLAP puro:**

- Velocidad queries agregadas: 100x más rápido
- Dashboard load time: 0.5s vs 30s

## 2.5.3.2.5.2.3: Row-Based vs Column-Based Storage

Diferencia Fundamental:

### ROW-BASED STORAGE:

ROW-BASED STORAGE (ROLAP tradicional)  
Usado en: SQL Server, Oracle, PostgreSQL

Tabla: fact\_sales

Estructura lógica:

| id | prod_id | cust_id | date    | amount |
|----|---------|---------|---------|--------|
| 1  | 101     | 201     | 2024-01 | 1000   |
| 2  | 102     | 201     | 2024-01 | 20     |
| 3  | 101     | 202     | 2024-01 | 1000   |
| 4  | 103     | 202     | 2024-02 | 50     |

Almacenamiento físico en disco:

PÁGINA 1 (8 KB):

Fila 1: [1][101][201][2024-01][1000]  
Fila 2: [2][102][201][2024-01][20]  
Fila 3: [3][101][202][2024-01][1000]  
Fila 4: [4][103][202][2024-02][50]

Bytes 0-24: Fila completa 1

Bytes 25-49: Fila completa 2

Bytes 50-74: Fila completa 3

Bytes 75-99: Fila completa 4

Características:

- Insertar fila: Rápido (append)
- Leer fila completa: Rápido (secuencial)
- Actualizar fila: Rápido (in-place)
- DELETE: Fácil (marcar como deleted)
- Leer UNA columna: LENTO (lee todas)
- Agregaciones: LENTO (lee todo)
- Compresión: Limitada (valores diversos)

## COLUMN-BASED STORAGE

COLUMN-BASED STORAGE (MOLAP, Data Warehouses)  
Usado en: MOLAP, Vertica, Redshift, BigQuery

Misma tabla lógica: fact\_sales

Almacenamiento físico en disco:

ARCHIVO: id.col

[1][2][3][4]

Todos los IDs juntos

ARCHIVO: product\_id.col

[101][102][101][103]

Todos los product\_ids juntos

ARCHIVO: customer\_id.col

[201][201][202][202]

ARCHIVO: date.col

[2024-01][2024-01][2024-01][2024-02]

ARCHIVO: amount.col

[1000][20][1000][50]

Características:

- Leer UNA columna: MUY RÁPIDO (solo esa)
- Agregaciones: MUY RÁPIDO (suma vector)
- Compresión: EXCELENTE (valores repetidos)
- Insertar fila: LENTO (5 archivos)
- Leer fila completa: LENTO (5 archivos)
- Actualizar valor: LENTO (reescribir)
- DELETE: Complejo (marcar en cada columna)

## Compresión en Column Store

EJEMPLO REAL: 1 millón de filas



Columna: customer\_id (row-based)

Valores: [201, 201, 201, 202, 202, 201, ...]

Sin compresión:

- 1,000,000 valores × 4 bytes (INT32)
- Total: 4,000,000 bytes (4 MB)

Con compresión general (gzip):

- Ratio: ~3:1
- Total: 1,333,333 bytes (1.3 MB)

Columna: customer\_id (column-based)

Valores: [201, 201, 201, 202, 202, 201, ...]

Dictionary Encoding:

Dictionary:

- 0 → 201
- 1 → 202
- 2 → 203
- ... (solo 5000 customers)

Encoded values:

[0, 0, 0, 1, 1, 0, 0, 1, ...]

Bits needed:  $\log_2(5000) = 13$  bits

Bytes per value: 2 bytes (rounded)

Dictionary size:

- 5000 entries × 4 bytes = 20 KB

Encoded data size:

- 1,000,000 × 2 bytes = 2 MB

TOTAL: 2.02 MB

+ Run-Length Encoding (RLE):

Si hay secuencias: [0,0,0,0,1,1,1,0,0]  
RLE: [(0,4), (1,3), (0,2)]

Compression ratio: ~10:1 típico

TOTAL FINAL: ~200 KB

Ahorro vs row-based: 4 MB → 200 KB (20x)

## Performance Comparison

BENCHMARK: Query de Agregación



Query: `SELECT SUM(amount) FROM fact_sales  
WHERE date >= '2024-01-01'`

Datos: 100 millones de filas

### ROW-BASED STORAGE

```
Paso 1: Leer datos
└── Lee TODAS las columnas de TODAS las filas
 └── Columnas: [id, prod_id, cust_id, date,
 amount, tax, discount, ...]
 └── Total: 8 columnas × 8 bytes = 64 bytes/fila
 └── 100M filas × 64 bytes = 6.4 GB

 └── Disco read:
 6.4 GB / 500 MB/s (SSD) = 12.8 segundos

 └── Filter date >= '2024-01-01':
 Lee TODAS las filas para filtrar
 Descarta 50M filas (2024-01-01 es mitad)

Paso 2: Sumar columna amount
└── Suma 50M valores
└── CPU time: 50M × 5 ns = 250 ms

TOTAL TIME: 12.8s + 0.25s = 13 segundos

I/O wasted: 75% (leyó 8 columnas, usó 2)
```

## CALCULO (Palabra clave)

### COLUMN-BASED STORAGE

```
Paso 1: Leer solo columnas necesarias
└── date.col (para filtro)
 └── 100M × 4 bytes = 400 MB
└── amount.col (para suma)
 └── 100M × 8 bytes = 800 MB

 └── Total a leer: 1.2 GB (vs 6.4 GB)

 └── Disco read:
 1.2 GB / 500 MB/s = 2.4 segundos

 └── BONUS: Compresión
 date.col comprimido: 100 MB (10:1 ratio)
 amount.col comprimido: 200 MB (4:1 ratio)
 Total: 300 MB
 Read time: 0.6 segundos

Paso 2: Filter + Sum (vectorizado)
└── Descomprimir: 100 ms
└── Filter SIMD: 50 ms
└── Sum SIMD: 25 ms

TOTAL TIME: 0.6s + 0.175s = 0.775 segundos

Speedup: 13s / 0.775s = 16.7x más rápido
```

## 2.5.3.2.5.2.4: X Query Routing en HOLAP

### Cómo HOLAP Decide Dónde Ejecutar Query

#### HOLAP QUERY ROUTER

---

##### PASO 1: Parse & Analyze

```
Extrae:
- Medida: Sales (amount)
- Dimensión: Products
- Filtro: Time = 2024 Q4
- Agregación: SUM por producto
- Drill-through: NO
```

##### PASO 2: Check Partition Storage

```
Consultar partition_storage_map.xml:

2024 Q4 → Partition "2024_current"
Storage mode: ROLAP
Connection: dwh.company.com
```

##### PASO 3: Check Aggregation Availability

```
Buscar en aggregation_storage_map.xml:

¿Existe by_product aggregation?
└─ YES: by_product.agg (MOLAP)
 Last updated: Today 2:00 AM
 Includes: All dates including Q4
 BUT: Updated nightly (stale data)

└─ Decision matrix:
 User wants real-time? → ROLAP
 User OK with 12h old? → MOLAP agg
```

## PASO 4: Choose Execution Path

### OPTION A: Use MOLAP aggregation

#### Pros:

- Muy rápido (0.1s)
- No carga DWH

#### Cons:

- Datos de anoche (stale 14 horas)
- Falta ventas de hoy

### OPTION B: Query ROLAP directly

#### Pros:

- Datos actualizados (real-time)
- Incluye ventas de hoy

#### Cons:

- Más lento (3-5s)
- Carga el DWH

### OPTION C: Hybrid (MEJOR)

1. Leer MOLAP agg hasta ayer  
(2024-10-01 hasta 2024-12-28)  
Time: 0.05s

2. Query ROLAP para HOY  
(2024-12-29)  
Time: 0.5s

3. Merge results  
Time: 0.01s

TOTAL: 0.56s

#### Pros:

- Datos actualizados
- Relativamente rápido
- Carga mínima en DWH

DECISION: OPTION C (Hybrid)

## PASO 5: Execute Hybrid Plan

### Thread 1: Read MOLAP

- Load by\_product.agg
- Filter to 2024 Q4 (Oct-Dec)
- Exclude today (Dec 29)

### Thread 2: Query ROLAP (parallel)

- Generate SQL:

```
SELECT
 p.product_name,
 SUM(f.amount)
FROM fact_sales f
JOIN dim_products p
 ON f.product_id = p.id
WHERE f.date = '2024-12-29'
GROUP BY p.product_name
```
- Execute on DWH

### Merge:

- Combine results by product
- MOLAP: Product A = 100K
- ROLAP: Product A = 5K (today)
- Final: Product A = 105K

## **2.5.3.2.5.2.4.1: Ejemplo de Decisión Real**

### **QUERY TYPE 1: Agregación Simple**

Query: "Total sales 2024"

Analysis:

- Medida: SUM(Sales)
- Dimensión: None (grand total)
- Filtro: Year 2024
- Detalle: NO

Routing Decision:

- Use MOLAP: grand\_total.agg
  - └ Filter to 2024 partition
  - └ Time: 0.001s

Reason: Agregación perfecta, pre-calculada

---

### **QUERY TYPE 2: Drill-Down con Filtro**

Query: "Sales by product where brand='Dell'  
and price > 1000"

Analysis:

- Medida: SUM(Sales)
- Dimensión: Products
- Filtro: Brand, Price (attributes)
- Detalle: NO

Routing Decision:

- ⚠ PARTIAL MOLAP:
  1. Use by\_product.agg (MOLAP)
  2. Post-filter in memory:
    - Join with products dimension store
    - Filter brand='Dell' AND price>1000
  3. Return filtered results

Time: 0.1s

Reason: Agg útil pero necesita post-filter

---

## QUERY TYPE 3: Drill-Through a Detalle

Query: "Show all transactions for Product 123  
in December 2024"

Analysis:

- Medida: N/A (raw data)
- Dimensión: N/A
- Filtro: Product, Date
- Detalle: YES (transaction level)

Routing Decision:

- CANNOT use MOLAP (no detail)  
 MUST use ROLAP:

Generate SQL:

```
SELECT *
FROM fact_sales f
WHERE f.product_id = 123
 AND f.date BETWEEN '2024-12-01'
 AND '2024-12-31'
ORDER BY f.date, f.time
```

Time: 2-5s

Reason: Necesita filas individuales

---

## **QUERY TYPE 4: Complex Ad-Hoc**

Query: "Customers who bought Product A  
but not Product B, sorted by  
total lifetime value"

Analysis:

- Medida: Multiple (exists, not exists, sum)
- Dimensión: Customers
- Filtro: Complex logic
- Detalle: Intermediate

Routing Decision:

- IMPOSSIBLE in MOLAP  
 MUST use ROLAP:

Generate SQL:

```
WITH bought_a AS (
 SELECT DISTINCT customer_id
 FROM fact_sales
 WHERE product_id = 'A'
),
bought_b AS (
 SELECT DISTINCT customer_id
 FROM fact_sales
 WHERE product_id = 'B'
),
lifetime_value AS (
 SELECT
 customer_id,
 SUM(amount) as ltv
 FROM fact_sales
 GROUP BY customer_id
)
SELECT
 c.customer_name,
 lv.ltv
FROM bought_a ba
LEFT JOIN bought_b bb
 ON ba.customer_id = bb.customer_id
JOIN lifetime_value lv
 ON ba.customer_id = lv.customer_id
JOIN dim_customers c
 ON ba.customer_id = c.id
WHERE bb.customer_id IS NULL
ORDER BY lv.ltv DESC
```

Time: 10-30s

Reason: Complejidad requiere SQL completo

## **QUERY TYPE 5: Time Intelligence**

Query: "Sales YTD vs Prior Year YTD"

Analysis:

- Medida: SUM(Sales)
- Dimensión: Time
- Filtro: YTD calculation
- Detalle: NO

Routing Decision:

Use MOLAP with MDX:

```
SELECT
 {[Measures].[Sales]} ON 0,
 {
 PeriodsToDate([Time].[Year],
 [Time].CurrentMember),
 ParallelPeriod([Time].[Year], 1,
 [Time].CurrentMember)
 } ON 1
FROM [Sales]
```

Time: 0.05s

Reason: Time intelligence built into MOLAP

## 2.5.3.2.5.2.5: Convergencia de Sistemas Modernos

### Cómo los Sistemas Actuales Combinan Todo

---

#### TENDENCIA 1: Columnar Everywhere

Sistemas tradicionales ROLAP adoptando columnar storage:

SQL Server:

- └ Row store (default)
- └ + Columnstore indexes (optional)
  - └ "CREATE COLUMNSTORE INDEX ..."

PostgreSQL:

- └ Row store (heap)
- └ + cstore\_fdw extension (columnar)

Oracle:

- └ Row store
- └ + In-Memory Column Store

Resultado:

ROLAP con performance de MOLAP para agregaciones

#### TENDENCIA 2: In-Memory Computing

OLTP + OLAP en RAM:

SAP HANA:

- └ Row store (OLTP operations)
- └ Column store (Analytics)
- └ TODO en RAM (TB de memoria)
- └ Híbrido automático

SQL Server In-Memory OLTP:

- └ Memory-optimized tables
- └ + Columnstore on memory-optimized

Resultado:

Queries OLAP en microsegundos

## TENDENCIA 3: Cloud Data Warehouses

Snowflake:

- Columnar storage (automático)
- Separación compute/storage
- Auto-scaling
- Materializ views = MOLAP aggs

BigQuery:

- Columnar storage
- Distributed processing
- BI Engine = In-memory MOLAP layer

Redshift:

- Columnar (C-store)
- Distribution keys
- Materialized views

Resultado:

SQL queries con velocidad MOLAP  
Sin necesidad de cubo separado

## TENDENCIA 4: Lakehouse Architecture

Delta Lake / Iceberg:

Data Lake (Parquet files)

- Columnar format
- Partitioning
- Statistics

+ Transaction Layer

- ACID guarantees
- Time travel
- Schema evolution

+ Query Engine

- Spark / Presto / Trino
- Vectorized execution
- Predicate pushdown

Resultado:

Data Lake con capacidades de DWH  
Esquema flexible + Performance MOLAP

## TENDENCIA 5: Semantic Layer Universal

Power BI / Tableau:

- Semantic Model (universal)
  - Define dimensions/measures once
  - Works with ANY backend:
    - MOLAP (VertiPaq)
      - In-memory columnar
    - ROLAP (DirectQuery)
      - SQL pushdown
    - HOLAP (Hybrid)
      - Aggs in-memory, detail in SQL
  - Automatic query routing

Looker / dbt:

- Metrics layer (LookML / YAML)
- Compiles to SQL
- Works on any SQL DWH

Resultado:

Abstracción total del storage  
Usuario no sabe si es ROLAP/MOLAP/HOLAP

# El Sistema Ideal Moderno (2024)

## ARQUITECTURA MODERNA CONVERGENTE

### LAYER 1: Data Storage (Multi-format)

- Raw Data Lake (Parquet/Delta)
  - Columnar format
  - Partitioned by date
  - Schema on read
  
- Data Warehouse (Snowflake/BigQuery)
  - Columnar storage
  - Optimized for analytics
  - Materialized views (=MOLAP aggs)
  
- In-Memory Cache
  - Hot aggregations
  - Recent detail data

### LAYER 2: Query Engine (Intelligent Routing)

- Query Optimizer decides:
  
- Aggregation query?
  - Check materialized views
    - Fresh? Use it (0.1s)
    - Stale? Compute (5s)
  
- Detail query?
  - Check cache
    - Hit? Return (0.01s)
    - Miss? Query DWH (2s)
  
- Complex ad-hoc?
  - Full SQL on DWH (10s)

### LAYER 3: Semantic Layer (Business Logic)

- Metrics Repository
  - Sales = SUM(amount)
  - Profit = Sales - Cost
  - YoY\_Growth = (This - Last) / Last
  - ... (100s of metrics)
  
- Automatic Translation:
  - MDX (for MOLAP cubes)
  - SQL (for DWH)
  - DAX (for Power BI)

#### LAYER 4: Presentation (User Interface)

- Dashboards (Power BI/Tableau)
  - Real-time tiles (MOLAP-like)
  - Drill-through (ROLAP-like)
  - Ad-hoc (SQL-like)

- Reports (scheduled)
  - Pre-computed overnight
  - Served from cache

- Notebooks (exploratory)
  - Full SQL/Python access

## 2.5.3.2.5.2.6: Resumen Ejecutivo

### ROLAP vs MOLAP vs HOLAP

| Aspecto            | ROLAP                                                                                       | MOLAP                                                                                      | HOLAP                                                                                   |
|--------------------|---------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| Storage            | Relational DB                                                                               | Multidimensional arrays                                                                    | Híbrido                                                                                 |
| Format             | Row-based (típico)                                                                          | Column-based                                                                               | Ambos                                                                                   |
| Queries agregadas  | 5-30s                                                                                       | 0.01-0.1s                                                                                  | 0.1-1s                                                                                  |
| Queries detalladas | 2-10s                                                                                       | Imposible/lento                                                                            | 2-10s                                                                                   |
| Ad-hoc queries     |  Excelente |  Limitado |  Bueno |
| Actualización      | Tiempo real                                                                                 | Batch (lento)                                                                              | Híbrido                                                                                 |
| Tamaño disco       | Base                                                                                        | 5-10x base                                                                                 | 2-4x base                                                                               |
| Complejidad        | Baja                                                                                        | Media                                                                                      | Alta                                                                                    |

### Cuándo Usar Cada Uno

#### ROLAP:

- Datos cambian constantemente
- Necesitas queries ad-hoc
- Drill-through frecuente
- Presupuesto limitado

#### MOLAP:

- Datos históricos estables
- Dashboard performance crítico
- Queries predecibles
- Storage no es problema

#### HOLAP:

- Mejor de ambos mundos
- Datos recientes (ROLAP) + históricos (MOLAP)
- Balance costo/performance
- Empresa mediana/grande

### Convergencia Moderna

Los sistemas modernos están convergiendo hacia:

1. **Columnar storage** para todo
2. **In-memory** para hot data
3. **Intelligent routing** automático
4. **Semantic layer** universal
5. **Cloud-native** scaling

Resultado: Ya no necesitas elegir ROLAP vs MOLAP. El sistema elige por ti, dinámicamente.

## 2.5.3.2.5.3: Drill-through vs Ad-hoc - ¿molap usa dax o mdx? – Consulta HOLAP

### Drill-through:

Dashboard → Click en "Laptop: \$150K"  
→ Ver las 245 transacciones individuales

- = Navegación desde agregado a DETALLE
- = Path predefinido (cubo → tabla raw)
- = Mismo contexto/filtros

### Ad-hoc:

Usuario escribe query libre:  
"Clientes que compraron A pero no B,  
ordenados por lifetime value"

- = Query NUEVA, no predefinida
- = Usuario crea la lógica
- = SQL/MDX completo

### Diferencia clave:

| Aspecto     | Drill-through            | Ad-hoc                  |
|-------------|--------------------------|-------------------------|
| Libertad    | Limitada (solo detalles) | Total (cualquier query) |
| Path        | Predefinido por cubo     | Usuario lo inventa      |
| Complejidad | Simple (click)           | Compleja (SQL)          |

### Ejemplo:

Drill-through:  
Dashboard: "Ventas por región: West \$500K"  
→ Click  
→ Tabla: 1,234 transacciones de West

Ad-hoc:  
Usuario: "SELECT customers que compraron  
en West PERO viven en East"  
→ Query nueva que el cubo no contempla

**En MOLAP:** Drill-through difícil, Ad-hoc imposible

**En ROLAP:** Ambos posibles

**En HOLAP:** Drill-through a ROLAP, Ad-hoc a ROLAP

## **MDX (tradicional) y DAX (moderno)**

### **MDX (Multidimensional Expressions)**

- Lenguaje ORIGINAL para MOLAP
- Usado en: SSAS Multidimensional, Oracle Essbase
- Sintaxis compleja, específico para cubos

### **DAX (Data Analysis Expressions)**

- Lenguaje MODERNO para MOLAP
- Usado en: Power BI, SSAS Tabular, Excel Power Pivot
- Sintaxis tipo Excel, más fácil

#### **Ejemplo:**



```
-- MDX (viejo)
SELECT
 {[Measures].[Sales]} ON COLUMNS,
 {[Products].[Category].Members} ON ROWS
FROM [SalesCube]

-- DAX (nuevo)
CALCULATE(
 SUM(Sales[Amount]),
 Products[Category] = "Electronics"
)
```

#### **Resumen:**

- **MDX** = MOLAP clásico (cubos multidimensionales)
- **DAX** = MOLAP moderno (tabular/columnar)
- Ambos son para MOLAP, pero generaciones diferentes

## Consulta HOLAP (invisible para el usuario)

El usuario NO ve diferencia:



-- Usuario escribe (DAX/MDX normal):

```
SELECT
 Product,
 SUM(Sales)
FROM SalesCube
WHERE Year = 2024
```

HOLAP decide internamente:

```
Query Router:
 2024 Q1-Q3 (histórico) → Lee MOLAP local
 Time: 0.05s

 2024 Q4 (actual) → Genera SQL a ROLAP:
 SELECT product_id, SUM(amount)
 FROM fact_sales
 WHERE date >= '2024-10-01'
 Time: 0.5s

 Merge ambos resultados → Usuario
 Total: 0.55s
```

El usuario ve:

| Product        |  | Sales   |
|----------------|--|---------|
| Laptop Dell    |  | 150,200 |
| Mouse Logitech |  | 3,450   |
| ...            |  |         |

**Clave:** Usuario usa MDX/DAX normal. HOLAP traduce automáticamente parte a SQL si necesita ROLAP. No existe "lenguaje HOLAP" - es transparente.

**holap siempre sql ? o puede usar tambien dax?**

HOLAP puede usar AMBOS

Depende de la tecnología:

**SQL Server Analysis Services (SSAS):**

Usuario escribe: MDX



HOLAP internamente:

- └ Partición MOLAP → Ejecuta MDX localmente
- └ Partición ROLAP → Traduce a SQL automáticamente

**Power BI (moderno):**

Usuario escribe: DAX

HOLAP internamente:

- └ Datos en memoria → Ejecuta DAX (VertiPaq)
- └ DirectQuery → Traduce a SQL automáticamente

**Ejemplo Power BI:**



```
// Usuario escribe DAX:
Total Sales =
CALCULATE(
 SUM(Sales[Amount]),
 FILTER(Products, Products[Category] = "Electronics")
)

// HOLAP decide:
// - Import mode (MOLAP) → Ejecuta DAX en memoria
// - DirectQuery (ROLAP) → Traduce a:
// SELECT SUM(amount) FROM sales
// WHERE category = 'Electronics'
```

**Resumen:**

- **Lenguaje del usuario:** MDX o DAX (según herramienta)
- **Ejecución interna HOLAP:**
  - MOLAP → MDX/DAX nativo
  - ROLAP → Traduce a SQL automáticamente

**El SQL es interno, el usuario no lo ve.**

---

super breve. osea que HOLAP puede usar tanto dax como sql

 Correcto

HOLAP usa AMBOS simultáneamente:



Ejemplo real:



```
// Usuario escribe DAX:
Total = SUM(Sales[Amount])

// HOLAP ejecuta:
// Histórico → DAX en memoria (MOLAP)
// Actual → SELECT SUM(amount) FROM... (SQL a ROLAP)
// Combina ambos
```

Resumen ultra breve:

- **Frontend:** DAX/MDX (lo que ve el usuario)
- **Backend HOLAP:** DAX + SQL (automático, invisible)

HOLAP = "Políglota" (habla ambos lenguajes internamente)

---

## **HOLAP puede exponer SQL directamente**

Depende del modo de acceso:

Modo 1: Via herramienta BI (DAX/MDX)

Power BI → DAX → HOLAP traduce a SQL si necesita  
SSAS → MDX → HOLAP traduce a SQL si necesita

Modo 2: SQL directo al cubo HOLAP



-- Usuario conecta con SQL a HOLAP:

```
SELECT
 p.name,
 SUM(f.amount)
FROM sales_cube f
JOIN products_dim p ON f.product_id = p.id
WHERE year = 2024
GROUP BY p.name
```

-- HOLAP decide:

-- Histórico → Ejecuta sobre MOLAP (optimizado)  
-- Actual → Pass-through a ROLAP (SQL nativo)

Herramientas que permiten SQL directo a HOLAP:

- Tableau - Looker - dbt - Cualquier cliente SQL estándar

**Resumen:**

**HOLAP soporta 3 lenguajes:**

1. DAX (Power BI)
2. MDX (SSAS clásico)
3. SQL (directo o traducido)

El usuario elige según su herramienta.

## 2.5.3.3: 3. ¡!!HOLAP - Representación Física (Híbrido) (retomando: nota: la mayoría de los conceptos que se explican en este apartado ya se explicaron en el 2.5.3.2)

Como se almacena: AGREGACIONES en cubo (array-based) + Detallada en tablas

⚠ IMPORTANTE: HOLAP combina AMBOS enfoques

Parte 1 (Agregaciones): MOLAP → Array-based multidimensional

Parte 2 (Detalles): ROLAP → Row-based O Column-based (según BD)

### Recordatorio: Ubicación de ROLAP vs MOLAP en HOLAP

#### HOLAP (Hybrid OLAP)

##### ROLAP (servidor remoto)

- Base de datos SQL (MySQL, PostgreSQL)
- Tablas relacionales
- En servidor/nube
- Datos recientes/actuales
- Drill-through y detalles

↔ Red

##### MOLAP (local/cliente)

- Archivos .dat locales
- Cubos pre-agregados
- En disco local o cache
- Datos históricos
- Queries rápidas

#### Resumen:

- **ROLAP:** Servidor SQL (fuente de verdad, datos frescos)
- **MOLAP:** Local/cache (cubo pre-procesados, lectura rápida)

### 2.5.3.3.1: Estructura hibrida HOLAP:

#### Estructura hibrida HOLAP

HOLAP System/

Aggregations (MOLAP - ARRAY-BASED)  
sales\_agg.mdb

Array multidimensional (solo totales)

```
AggArray[0][0]['ALL'] = 2000 <
AggArray[1][0]['ALL'] = 40
AggArray['ALL'][0][0] = 1020
AggArray['ALL'][0]['ALL'] = 2040
AggArray['ALL']['ALL']['ALL'] = 3090 |
```

Tamaño: 500 bytes

Storage: Array N-dimensional (como MOLAP)

Detail Data (ROLAP - puede ser ROW o COLUMN)

Opción A: ROW-BASED (PostgreSQL tradicional) -> **fact\_sales.dat**

```
Row 1: [1, 101, 201, 1000, ...]
Row 2: [2, 102, 201, 20, ...]
Row 3: [3, 101, 202, 1000, ...]
```

Tamaño: 600 bytes

Storage: Row-based secuencial

Opcion B: COLUMN-BASED  
(Redshift/BigQuery moderno)

```
fact_sales/ (directorio)
 column_id.dat: [1, 2, 3, 4, 5, 6]
 column_product.dat: [101, 102, 101, ...]
 column_customer.dat: [201, 201, 202, ...]
 column_amount.dat: [1000, 20, 1000, ...]
```

Tamaño: 600 bytes

Storage: Column-based separado

TOTAL TAMAÑO: 500 bytes (MOLAP) + 600 bytes (ROLAP) = 1.100 bytes

### **2.5.3.3.2: Decisión Inteligente de Query Engine:**

#### **Query 1: "¿Ventas totales de Alice?" (agregado simple)**

Engine analiza: Pregunta por total → Dato agregado

Decisión: Usar MOLAP (array-based)

Proceso:

1. Lookup en sales\_agg.mdb
2. Acceso array: AggArray['ALL'][Alice]['ALL']
3. Lee valor: 2040

Storage usado: Array-based multidimensional

Tiempo: 0.1 segundos ↗

#### **Query 2: "¿Qué compró Alice el 15 de Enero a las 3pm?" (detalle específico)**

Engine analiza: Pregunta por transacción específica → Dato detallado

Decisión: Usar ROLAP

Si Row-based:

1. Scan fact\_sales.dat
2. Filtrar customer=Alice AND date='2024-01-15' AND hour=15
3. Lee fila completa: [1, 101, 201, 1000, '2024-01-15 15:00']
4. Resultado: Laptop (\$1000)

Storage usado: Row-based secuencial

Tiempo: 2 segundos

Si Column-based:

1. Lee column\_customer.dat → encuentra índices [0, 1, 3]
2. Lee column\_date.dat → filtra por fecha
3. Lee column\_product.dat → obtiene producto
4. Resultado: Laptop (\$1000)

Storage usado: Column-based selectivo

Tiempo: 0.5 segundos

#### **Query 3: "¿Ventas de Laptop por mes en 2024?" (semi-agregado)**

Engine analiza: Agregado por mes -> puede estar pre-calculado

Decisión: Intentar MOLAP primero

Si existe en cubo:

1. Lookup: AggArray[Laptop]['ALL'][cada\_mes]
2. Lee valores: [Jan: 2000, Feb: 0, Mar: 1000]

Storage usado: Array-based

Tiempo: 0.2 segundos

Si NO existe en cubo (agregación no pre-calculada):

1. Fallback a ROLAP
2. Query desde fact\_sales (row o column según BD)
3. GROUP BY month

Storage usado: Row-based o Column-based

Tiempo: 1-3 segundos

### 2.5.3.3.3: Comparación Storage HOLAP y Ventajas de diseño híbrido

## HOLAP = Dos Storages

Agregaciones (MOLAP):

Row 1: [1, 101, 201, 1000, ...]  
Row 2: [2, 102, 201, 20, ...]  
Row 3: [3, 101, 202, 1000, ...]

↑ (query engine decide)

Detalles (ROLAP)

Row-based:  
[row1][row2][row3]  
OR  
[col1][col2][col3]...  
Tipo: Relacional (2D)

1. Queries agregadas → MOLAP (array-based, ultra rápido)  
"Total ventas por región" → 0.1 segundos
2. Queries detalladas → ROLAP (row/column-based, flexible)  
"Transacciones del cliente X" → 0.5-2 segundos
3. Balance perfecto:
  - Espacio: Menos que MOLAP puro
  - Velocidad: Más rápido que ROLAP puro
  - Flexibilidad: Queries ad-hoc posibles (ROLAP)

## 2.5.3.3.4: Ejemplo Real: Sistema HOLAP con Column-based ROLAP – Tabla Comparativa

### Completa: Row vs Column vs Array – Query HOLAP (inteligente)

|                                                                                                                                                                                         |  |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Empresa moderna usa:                                                                                                                                                                    |  |
| - Aggregations: Microsoft SSAS (MOLAP array-based)                                                                                                                                      |  |
| - Details: Amazon Redshift (ROLAP column-based)                                                                                                                                         |  |
| Setup:                                                                                                                                                                                  |  |
| Aggregations (SSAS MOLAP)<br>Array[product][customer][time]<br>5 mil millones celdas pre-calculadas<br>Tamaño: 500 GB<br>Storage: Multidimensional array                                |  |
| ↓                                                                                                                                                                                       |  |
| Details (Redshift ROLAP)<br>Column-based storage:<br>- column_product/<br>- column_customer/<br>- column_amount/<br>10 mil millones filas<br>Tamaño: 2 TB<br>Storage: Columnar separado |  |

Query común "Total ventas por región":

→ SSAS MOLAP (array) → 0.1 seg ⚡

Query rara "Listar todas las transacciones >\$10K de cliente específico":

→ Redshift ROLAP (columnar) → 1 seg

Mejor de ambos mundos

### Tabla Comparativa Completa: Row vs Column vs Array

| Aspecto           | ROLAP Row       | ROLAP Column       | MOLAP Array      | HOLAP           |
|-------------------|-----------------|--------------------|------------------|-----------------|
| Estructura básica | Filas juntas    | Columnas juntas    | Arrays N-D       | Ambos           |
| Storage type      | Row-based       | Column-based       | Array-based      | Array + Row/Col |
| Unidad de lectura | 1 fila completa | 1 columna completa | 1 celda (coords) | Depende         |
| Dimensionalidad   | 2D (tabla)      | 2D (tabla)         | N-D (cubo)       | 2D + N-D        |
| Pre-cálculo       | ✗ No            | ✗ No               | ✓ Sí (todo)      | ✓ Sí (parcial)  |
| Mejor para        | OLTP            | OLAP queries       | OLAP drill-down  | Híbrido         |
| Ejemplo tech      | PostgreSQL      | Redshift           | SSAS MOLAP       | SSAS HOLAP      |
| Query agregada    | 10-30 seg       | 1-5 seg            | 0.01-0.1 seg     | 0.1-0.5 seg     |
| Query detalle     | 0.5-2 seg       | 0.5-2 seg          | ✗ No disponible  | 0.5-2 seg       |
| Espacio disco     | ● Bajo          | ● Bajo             | ● Alto           | ● Medio         |
| Flexibilidad      | ✓ Alta          | ✓ Alta             | ✗ Baja           | ✓ Alta          |
| Compresión        | Baja            | Alta               | Muy alta         | Media-Alta      |

## **Query HOLAP (inteligente):**

### **Query 1: "¿Ventas totales de Alice?" (agregado)**

Sistema decide: Usar MOLAP (agregación guardada)

Proceso:

1. Lookup en sales\_agg.mdb
2. Lee: [ALL][Alice][ALL] = 2040

Tiempo: 0.1 segundos ↗

### **Query 2: "¿Qué compró Alice el 15 de Enero?" (detalle)**

Sistema decide: Usar ROLAP (detalle no pre-calculado)

Proceso:

1. Query a fact\_sales.dat
2. SELECT \* WHERE customer\_id = 201 AND date = '2024-01-15'
3. Resultado: Laptop (\$1000), Mouse (\$20)

Tiempo: 2 segundos

### **Query 3: "¿Ventas de Laptop por mes para Alice?" (semi-agregado)**

Sistema decide: Usar MOLAP (si existe agregación mensual)

Si NO existe: Usar ROLAP (calcular desde detalles)

Tiempo: 0.5-2 segundos (dependiendo)

#### 2.5.3.4: 4. CRÍTICO: Disco vs In-Memory - Concepto SEPARADO

⚠ IMPORTANTE: Dos Conceptos INDEPENDIENTES (Ortogonales)

Concepto 1: ESTRUCTURA de datos

- ROLAP (relacional - tablas)
- MOLAP (multidimensional - arrays)
- HOLAP (híbrido)



Concepto 2: UBICACIÓN física

- Disco (HDD/SSD)
- In-Memory (RAM)

Estos conceptos son INDEPENDIENTES

Puedes combinar CUALQUIERA con CUALQUIERA

#### 2.5.3.4.1: Matriz Completa: Estructura x Ubicación

|                           | Disco (tradicional)              | In-Memory(moderno)          |
|---------------------------|----------------------------------|-----------------------------|
| <b>ROLAP Row-Based</b>    | PostgreSQL, MySQL tradicional    | MemSQL, VoltDB              |
| <b>ROLAP Column-Based</b> | Redshift, BigQuery (disco+cache) | SAP HANA, Vertica in-memory |
| <b>MOLAP Array-Based</b>  | SSAS MOLAP tradicional           | SSAS MOLAP in-memory mode   |
| <b>HOLAP</b>              | SSAS HOLAP tradicional           | SSAS HOLAP con in-memory    |

Conclusión: Cualquier estructura (ROLAP/MOLAP/HOLAP) puede estar en disco O en memoria

#### 2.5.3.4.2: Ejemplo Detallados

##### 2.5.3.4.2.1: ROLAP en DISCO (Tradicional)

###### Ejemplo: PostgreSQL en disco duro

Ubicación física: /var/lib/postgresql/data/

fact\_sales.dat (en HDD/SSD):

- └─ Datos almacenados: DISCO
- └─ Query ejecuta: Lee desde DISCO → carga a RAM → procesa
- └─ Tiempo lectura: 100ms (disco) + cálculo

Flujo:

Usuario hace query

↓

PostgreSQL lee disco (100ms) 📁

↓

Carga datos en RAM temporalmente

↓

Calcula resultado

↓

Devuelve resultado

Tiempo total: 5-30 segundos

Almacenamiento permanente: DISCO

##### 2.5.3.4.2.2: ROLAP en IN-MEMORY (Moderno)

###### Ejemplo: SAP HANA (ROLAP column-based in-memory)

Ubicación física: TODO en RAM

Datos almacenados: RAM (no disco durante operación)

Query ejecuta: Lee directamente de RAM (0.1ms) ↘

Backup: Disco (solo para persistencia)

Flujo:

Datos cargados en RAM al inicio

↓

Usuario hace query

↓

SAP HANA lee RAM (0.1ms) ↘

↓

Calcula resultado (también en RAM)

↓

Devuelve resultado

Tiempo total: 0.5-2 segundos (1000x más rápido que disco)

Almacenamiento permanente: DISCO (backup)

Almacenamiento activo: RAM

### 2.5.3.4.2.3: MOLAP en DISCO (Tradicional)

#### Ejemplo: SSAS MOLAP tradicional

Ubicación física: /opt/ssas/cubes/



sales\_cube.mdb (en HDD/SSD):

- └─ Datos pre-calculados: DISCO
- └─ Query ejecuta: Lee desde DISCO → devuelve valor
- └─ Tiempo lectura: 10-50ms (disco)

Flujo:

Usuario hace query "Total Alice"

↓

SSAS busca en disco (10-50ms)

↓

Lee valor pre-calculado: 2040

↓

Devuelve resultado

Tiempo total: 0.1-1 segundo

Almacenamiento: DISCO

### 2.5.3.4.2.4: MOLAP en IN-MEMORY (Menos común pero existe)

#### Ejemplo: SSAS MOLAP con DirectQuery mode in-memory

Ubicación física: Cubo cargado en RAM



sales\_cube.mdb cargado en RAM:

- └─ Datos pre-calculados: RAM
- └─ Query ejecuta: Lee desde RAM
- └─ Tiempo lectura: 0.01ms (RAM)

Flujo:

Cubo cargado en RAM al inicio

↓

Usuario hace query "Total Alice"

↓

SSAS lee RAM (0.01ms) ↗

↓

Devuelve valor pre-calculado: 2040

Tiempo total: 0.01-0.1 segundos (ultra rápido)

Almacenamiento permanente: DISCO (backup)

Almacenamiento activo: RAM

#### **2.5.3.4.2.5: HOLAP (siempre combinación)**

Ejemplo típico: SSAS HOLAP



Parte 1 (Agregaciones - MOLAP): Puede estar en DISCO o RAM

Parte 2 (Detalles - ROLAP): Puede estar en DISCO o RAM

Configuración común:

- └─ Agregaciones: MOLAP in-memory (RAM)
- └─ Detalles: ROLAP en disco (HDD/SSD)

Configuración alto rendimiento:

- └─ Agregaciones: MOLAP in-memory (RAM)
- └─ Detalles: ROLAP in-memory (RAM también)

## 2.5.3.4.3: ¿Dónde se Guarda Cada Uno Tradicionalmente?

### Histórico (1990s-2000s):

TODO en DISCO:

- └ ROLAP: PostgreSQL/Oracle en HDD
- └ MOLAP: SSAS MOLAP en HDD
- └ HOLAP: SSAS HOLAP en HDD

Razón: RAM era MUY cara (128MB costaba \$500)

### Moderno (2010s-2024):

HÍBRIDO (Disco + In-Memory):

ROLAP:

- └ Datos en disco (permanente)
- └ Cache en RAM (automático por BD)
- └ 0 totalmente in-memory (SAP HANA, MemSQL)

MOLAP:

- └ Datos pre-calculados en disco (permanente)
- └ Cubo cargado en RAM cuando se usa
- └ 0 totalmente in-memory (menos común)

HOLAP:

- └ Agregaciones: Preferible in-memory
- └ Detalles: Puede ser disco (más barato)

Razón: RAM más barata, necesidad de velocidad

## 2.5.3.4.4: Comparación: Storage + Location: toda la matriz de comparaciones

Escenario: 1M transacciones, Query "Total ventas Alice"

| Configuración                  | Tiempo       | Costo (estimado)     |
|--------------------------------|--------------|----------------------|
| ROLAP Row-Based en Disco       | 20-30 seg    | \$100 (1TB HDD)      |
| ROLAP Row-Based In-Memory      | 2-5 seg      | \$5,000 (64GB RAM)   |
| ROLAP Column-Based en Disco    | 3-5 seg      | \$200 (1TB SSD)      |
| ROLAP Column-Based In-Memory   | 0.5-1 seg    | \$5,000 (64GB RAM)   |
| MOLAP Array en Disco           | 0.5-1 seg    | \$500 (cubo 200GB)   |
| MOLAP Array In-Memory          | 0.01-0.1 seg | \$10,000 (200GB RAM) |
| HOLAP (Agg RAM + Detail Disco) | 0.1-0.5 seg  | \$3,000 (híbrido)    |

## 2.5.3.4.5: 📈 Decisión: ¿Disco o In-Memory?

Factores a considerar:

### 1. Tamaño de Datos

Datos < 100GB:

- In-Memory viable
- Costo razonable (~\$5,000 en RAM)

Datos 100GB - 1TB:

- Híbrido (agregados in-memory, detalles disco)
- Costo moderado

Datos > 1TB:

- Disco obligatorio
- In-memory solo para subset crítico

### 2. Frecuencia de Queries

Queries cada segundo (dashboards en vivo):

- In-Memory necesario

Queries cada minuto/hora:

- Disco con buen cache suficiente

Queries diarios (reportes batch):

- Disco totalmente OK

### 3. Presupuesto

Presupuesto alto:

- MOLAP in-memory (0.01 seg, \$10K+)

Presupuesto medio:

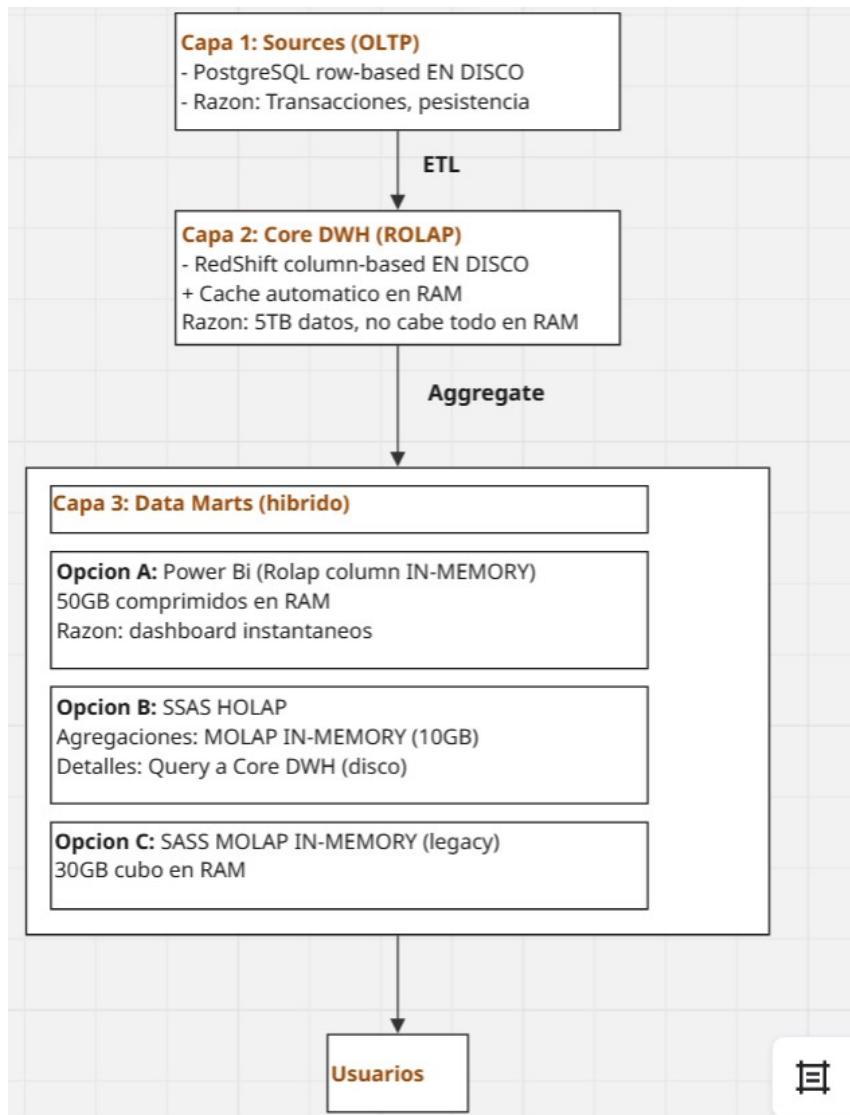
- HOLAP (agg RAM, detail disco) (0.1 seg, \$3K)
- O ROLAP column in-memory (0.5 seg, \$5K)

Presupuesto bajo:

- ROLAP column en disco (3 seg, \$200)

## 2.5.3.4.6: Arquitectura Real Moderna

Empresa típica 2024:



## 2.5.3.4.6.1:💡 Ejemplos Prácticos + Tabla Resumen Final

### Ejemplo 1: Startup (Presupuesto bajo)

Setup:

- Core DWH: PostgreSQL en disco (100GB)
- BI Tool: Power BI import (datos en RAM automático)

Costo:

- Servidor: \$200/mes
- Power BI Pro: \$10/usuario/mes

Performance:

- Queries Core: 5-10 segundos
- Queries Power BI: 0.5 segundos (in-memory automático)

Resultado: Balance perfecto para startup

### Ejemplo 2: Empresa Media (Presupuesto medio)

Setup:

- Core DWH: Redshift column-based disco (1TB)
- Data Marts: SSAS Tabular in-memory (50GB)
- Hot data: Últimos 3 meses en RAM

Costo:

- Redshift: \$1,000/mes
- Server in-memory: \$5,000 inicial + \$500/mes
- Total: ~\$1,500/mes

Performance:

- Queries históricas (Redshift disco): 2-5 seg
- Queries recientes (SSAS RAM): 0.1-0.5 seg

Resultado: 80% queries rápidas, 20% aceptables

### Ejemplo 3: Enterprise (Presupuesto alto)

Setup:

- Core DWH: Oracle Exadata (disco ultrarrápido) (10TB)
- Hot Data: SAP HANA in-memory column (500GB)
- OLAP Cubes: SSAS MOLAP in-memory (100GB)
- Todo redundante y distribuido

Costo:

- Hardware: \$500K inicial
- Licencias: \$100K/año
- Mantenimiento: \$50K/año
- Total: ~\$650K primer año

Performance:

- Queries históricas (Exadata): 0.5-2 seg
- Queries hot data (HANA): 0.1-0.5 seg
- Queries agregadas (SSAS): 0.01-0.1 seg

Resultado: TODO ultra rápido, alta disponibilidad

## Tabla Resumen Final

| Aspecto           | DISCO                                          | IN-MEMORY                                                     |
|-------------------|------------------------------------------------|---------------------------------------------------------------|
| Ubicación         | HDD/SSD                                        | RAM                                                           |
| Velocidad lectura | 10-100ms                                       | 0.01-0.1ms                                                    |
| Costo (1TB)       | \$100-500                                      | \$50,000-100,000                                              |
| Persistencia      | <input checked="" type="checkbox"/> Permanente | <input checked="" type="checkbox"/> Volátil (requiere backup) |
| Tamaño máximo     | Ilimitado                                      | Limitado a RAM disponible                                     |
| Mejor para        | Datos grandes/históricos                       | Datos recientes/hot                                           |
| ROLAP compatible  | <input checked="" type="checkbox"/> Sí         | <input checked="" type="checkbox"/> Sí                        |
| MOLAP compatible  | <input checked="" type="checkbox"/> Sí         | <input checked="" type="checkbox"/> Sí                        |
| HOLAP compatible  | <input checked="" type="checkbox"/> Sí         | <input checked="" type="checkbox"/> Sí (parcialmente)         |

### 2.5.3.4.6.2: Respuesta Directa LA Pregunta

"¿ROLAP/MOLAP/HOLAP se guarda en disco o in-memory?"

Respuesta: AMBOS – depende de la implementación

ROLAP puede estar:

- En disco (PostgreSQL tradicional)
- En memoria (SAP HANA, MemSQL)



MOLAP puede estar:

- En disco (SSAS MOLAP tradicional)
- En memoria (SSAS MOLAP in-memory mode)

HOLAP puede estar:

- Agregaciones en memoria + Detalles en disco (común)
- Todo en disco (tradicional)
- Todo en memoria (menos común, caro)

### Son decisiones SEPARADAS:

1. Elegir estructura: ROLAP/MOLAP/HOLAP
2. Elegir ubicación: Disco/In-Memory/Híbrido

## 2.5.3.4.6.3: ROLAP Columnar vs MOLAP – Diferencias

### Estructura de Almacenamiento

#### ROLAP COLUMNAR (ej: Parquet, ClickHouse)

Archivo por columna:

```
product_id.col: [5, 5, 5, 3, 8, 12, ...]
customer_id.col: [10, 10, 45, 10, 89, ...]
date_id.col: [0, 1, 15, 0, 5, ...]
amount.col: [1200, 0, 1200, 25, 150, ...]
quantity.col: [1, 0, 1, 1, 3, ...]
```

Orientado a COLUMNAS (vertical)

Cada columna = archivo separado

Lee solo columnas necesarias

#### MOLAP (Facts Array)

Array multidimensional:

```
sales_amount.dat:
[5][10][0] = 1200
[5][10][1] = 0
[5][45][15] = 1200
[3][10][0] = 25
...
```

Orientado a CELDAS (multidimensional)

Acceso por coordenadas [P][C][D]

Lee bloques de celdas

### Diferencias clave:

| Aspecto    | ROLAP Columnar         | MOLAP                      |
|------------|------------------------|----------------------------|
| Estructura | Columnas separadas     | Array N-dimensional        |
| Acceso     | Lee columnas completas | Lee celdas por coordenadas |
| Formato    | Archivo por columna    | Archivo por medida         |
| Indexación | IDs de negocio         | Posiciones (0,1,2...)      |
| Sparsity   | Guarda todas las filas | Solo celdas ocupadas       |
| Query      | Escanea columnas       | Calcula offsets            |

## Ejemplo: Query "SUM(amount) WHERE product=5"

### ROLAP COLUMNAR:

1. Leer product\_id.col completo  
[5, 5, 5, 3, 8, 12, 5, ...]
2. Filtrar posiciones donde product\_id=5  
Posiciones: [0, 1, 2, 6, ...]
3. Leer amount.col en esas posiciones  
amount[0] = 1200  
amount[1] = 0  
amount[2] = 1200  
amount[6] = 1200
4. Sumar: 3600

Operaciones: 2 lecturas de columnas

### MOLAP:

1. Ya sabes que producto 5 = posición 5
2. Leer directamente:  
FOR cliente in [0..4999]:  
 FOR día in [0..364]:  
 valor = sales\_amount[5][cliente][día]  
 sumar += valor
3. Total: 3600

Operaciones: Lecturas directas por offset

## Compresión

### ROLAP COLUMNAR:

- Run-Length Encoding (RLE)
- Dictionary encoding
- Comprime valores repetidos en columna

Ejemplo:

product\_id: [5,5,5,5,3,3,8,8,8]

Comprimido: [5x4, 3x2, 8x3]

## MOLAP

- Compresión por sparsity (celdas vacías)
- Dictionary en dimensions
- RLE en bloques de facts

### Array sparse:

[5][10][0] = 1200 ← guardado  
[5][10][1] = 0 ← no guardado (vacío)  
[5][10][2] = 0 ← no guardado

## Resumen

### ROLAP Columnar:

- Orientado a COLUMNAS
- Lee verticalmente
- Bueno para: "Dame toda la columna amount"

### MOLAP:

- Orientado a CELDAS/COORDENADAS
- Lee por offsets calculados
- Bueno para: "Dame [producto = 5] [cualquier cliente] [cualquier día]"

### RLE y Dictionary Encoding se usan en AMBOS

#### ROLAP Columnar

- MUY común (diseñado para esto)
- Parquet usa dictionary + RLE
  - ORC usa dictionary + RLE
  - ClickHouse usa dictionary + RLE

#### MOLAP

- También se usa (pero diferente)
- Dictionary en dimension stores
  - RLE en bloques de facts arrays
  - Compresión por sparsity (celdas vacías)

#### Diferencia:

- **ROLAP columnar:** Comprime COLUMNAS completas
- **MOLAP:** Comprime bloques/segmentos de arrays

**Las técnicas son las mismas, pero aplicadas a estructuras diferentes**

#### **2.5.3.4.6.4: Respuesta Directa a tu Pregunta ¿Disco o in-memory?**

"¿ROLAP/MOLAP/HOLAP se guarda en disco o in-memory?"

Respuesta: AMBOS - depende de la implementación.

ROLAP puede estar:

- └─ En disco (PostgreSQL tradicional)
- └─ En memoria (SAP HANA, MemSQL)

MOLAP puede estar:

- └─ En disco (SSAS MOLAP tradicional)
- └─ En memoria (SSAS MOLAP in-memory mode)

HOLAP puede estar:

- └─ Agregaciones en memoria + Detalles en disco (común)
- └─ Todo en disco (tradicional)
- └─ Todo en memoria (menos común, caro)

**Son decisiones SEPARADAS:**

1. Elegir estructura: ROLAP/MOLAP/HOLAP
2. Elegir ubicación: Disco/In-Memory/Híbrido

## 2.5.3.4.7: Storage Location: DISCO vs IN-MEMORY (RAM) repitiendo sección 2.5.3.4.5

**⚠️ IMPORTANTE:** Row/Column/Array se refiere a CÓMO se organizan los datos, NO a DÓNDE se guardan

Row-based / Column-based / Array-based = ORGANIZACIÓN  
(estructura)

Disco / In-Memory = UBICACIÓN (hardware)

Puedes tener:

- Row-based en DISCO (PostgreSQL tradicional)
- Row-based en RAM (PostgreSQL con cache grande)
- Column-based en DISCO (Redshift en disco)
- Column-based en RAM (SAP HANA, Power BI)
- Array-based en DISCO (SSAS MOLAP tradicional)
- Array-based en RAM (SSAS Tabular in-memory)

## 2.5.3.4.7.1: SUPER COMPLETO: ↑ Ubicación Real de ROLAP, MOLAP, HOLAP

### 2.5.3.4.7.1.1: └ ROLAP - Típicamente en DISCO

#### ROLAP Tradicional (Row-Based) – DISCO:

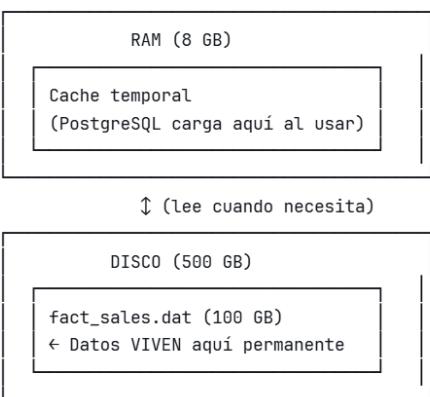
Tecnologías: PostgreSQL, MySQL, SQL Server, Oracle

Storage Location: DISCO (Hard Drive o SSD)

```
/var/lib/postgresql/data/
└ fact_sales.dat
 └ Guardado en DISCO permanentemente
```

Proceso de Query:

1. Query llega: "SELECT SUM(amount) FROM sales"
2. PostgreSQL LEE desde DISCO → carga a RAM temporalmente
3. Procesa en RAM
4. Devuelve resultado
5. Libera RAM (datos vuelven a estar solo en disco)



Ventaja: Barato (disco barato)

Desventaja: Lento (tiempo leer disco: 5-50ms)

#### ROLAP Moderno (Column-Based) - DISCO o HÍBRIDO:

Technologies: Amazon Redshift, GoogleBigQuery, Snowflake

Storage Location: PRINCIPALMENTE DISCO (cloud storage)  
+ Cache inteligente en RAM

Cloud Storage (S3):

```
column_amount.dat (comprimido)
column_product.dat (comprimido)
← Datos viven aquí permanentemente
```

↓

Server con RAM (cache):

```
RAM: Cache de columnas frecuentes
- column_amount (en RAM si se usa mucho)
- column_date (en RAM si se usa mucho)
```

Query frecuente: Responde desde RAM cache (rápido)

Query rara: Lee desde disco S3 (más lento)

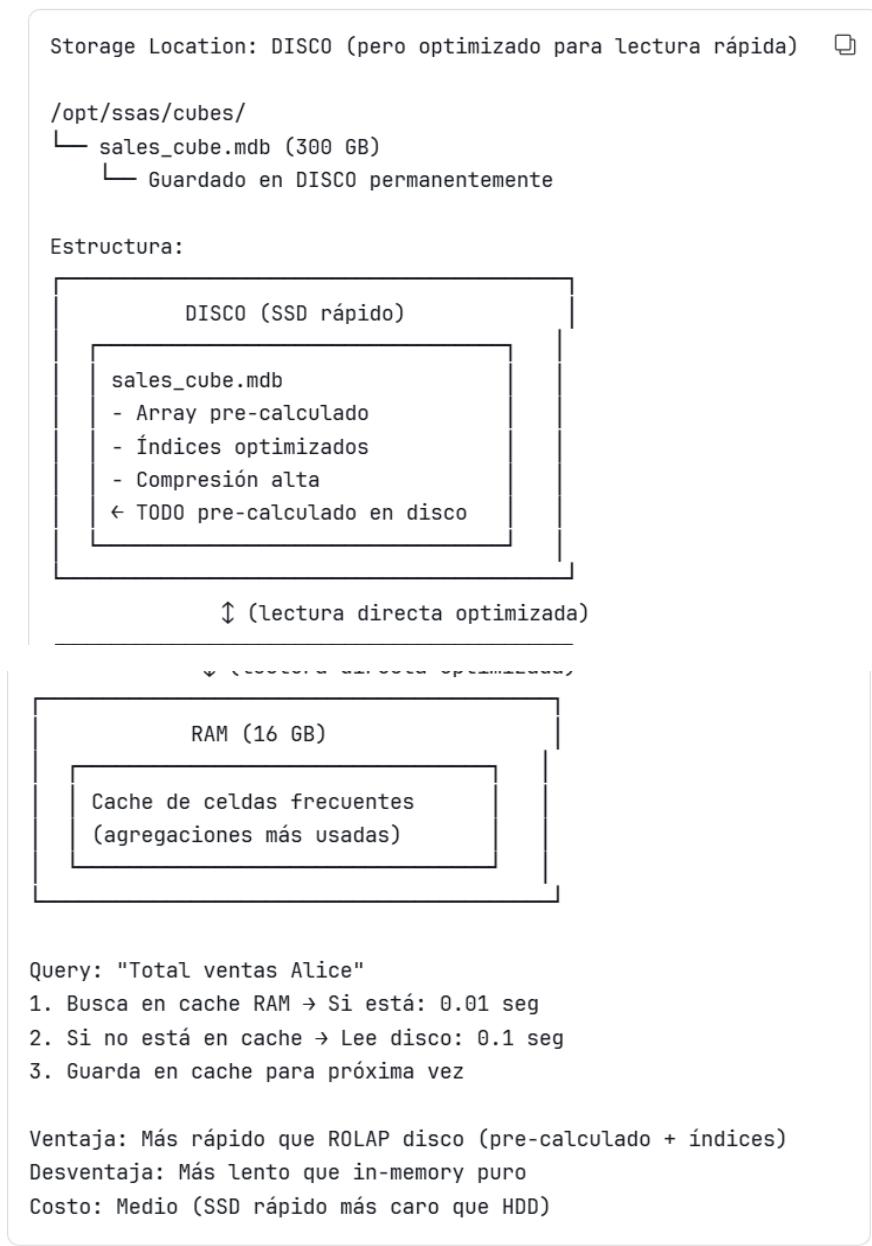
Ventaja: Balance costo/velocidad

Desventaja: No tan rápido como in-memory puro

## 2.5.3.4.7.1.2: ¿MOLAP - PUEDE SER DISCO o IN-MEMORY

### MOLAP Tradicional (Array-Based) - DISCO:

Tecnologías: Microsoft SSAS MOLAP clásico, IBM Cognos TM1



## MOLAP Moderno (In-Memory) - RAM:

Tecnologías: SAP HANA, Microsoft SSAS Tabular, Power BI (VertiPaq)

Storage Location: TODO EN RAM (memoria)



Setup inicial:

1. Datos extraídos del DWH
2. Construir cubo/modelo tabular
3. CARGAR TODO A RAM
4. Datos "viven" en RAM para queries

RAM (256 GB)

- Cubo COMPLETO aquí
- Todas las agregaciones
  - Todos los detalles
  - Columnar comprimido
  - ← TODO en RAM permanentemente

⇓ (solo para backup/snapshots)

DISCO (para persistencia)

- Snapshot cada 1 hora  
(backup si se cae server)

Query: "Total ventas Alice"

1. Lee directamente desde RAM: 0.001-0.01 seg
2. Sin acceso a disco

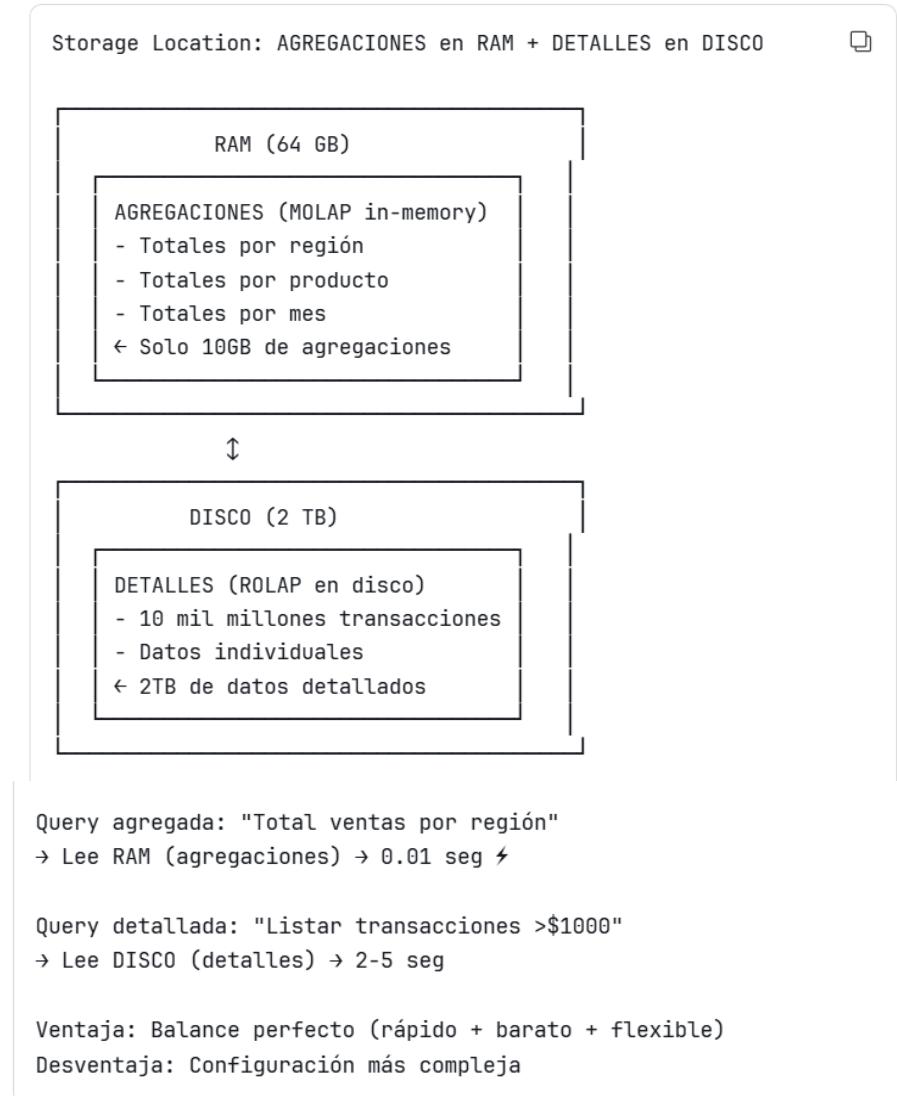
Ventaja: ULTRA RÁPIDO (RAM: 0.1ns, Disco: 5ms = 50,000x más lento)

Desventaja: MUY CARO (RAM: \$5-10/GB, Disco: \$0.10/GB)

Requisito: Datos deben caber en RAM (típicamente <500GB)

## 2.5.3.4.7.1.3: ⚡ HOLAP - HÍBRIDO DISCO + IN-MEMORY

Tecnologías: Microsoft SSAS HOLAP mode



## 2.5.3.4.7.2: 📊 Tabla Comparativa: Disco vs In-Memory

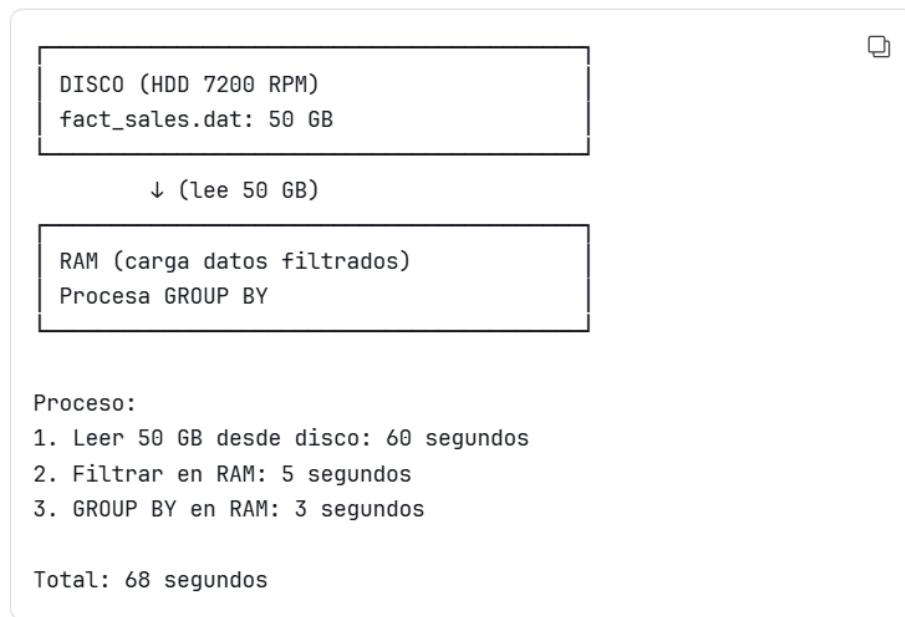
| Tecnología   | Organización | Ubicación Principal | Velocidad    | Costo    | Persistencia    |
|--------------|--------------|---------------------|--------------|----------|-----------------|
| PostgreSQL   | Row-based    | DISCO               | 5-30 seg     | 💡 Bajo   | ✓ Nativa        |
| MySQL        | Row-based    | DISCO               | 5-30 seg     | 💡 Bajo   | ✓ Nativa        |
| Redshift     | Column-based | DISCO + cache       | 1-5 seg      | 💡💡 Medio | ✓ Nativa        |
| BigQuery     | Column-based | DISCO (S3)          | 1-5 seg      | 💡💡 Medio | ✓ Nativa        |
| Snowflake    | Column-based | DISCO (S3)          | 1-5 seg      | 💡💡 Medio | ✓ Nativa        |
| SSAS MOLAP   | Array-based  | DISCO (SSD)         | 0.1-1 seg    | 💡💡 Medio | ✓ Nativa        |
| SAP HANA     | Column-based | IN-MEMORY (RAM)     | 0.01-0.1 seg | 💡💡💡 Alto | ⚠ Snapshots     |
| Power BI     | Column-based | IN-MEMORY (RAM)     | 0.01-0.1 seg | 💡💡 Medio | ⚠ Archivo .pbix |
| SSAS Tabular | Column-based | IN-MEMORY (RAM)     | 0.01-0.1 seg | 💡💡💡 Alto | ⚠ Snapshots     |
| HOLAP        | Híbrido      | DISCO + RAM         | 0.1-2 seg    | 💡💡 Medio | ✓ Parcial       |

### 2.5.3.4.7.3: 🔎 Ejemplo Real: Query en Diferentes Ubicaciones

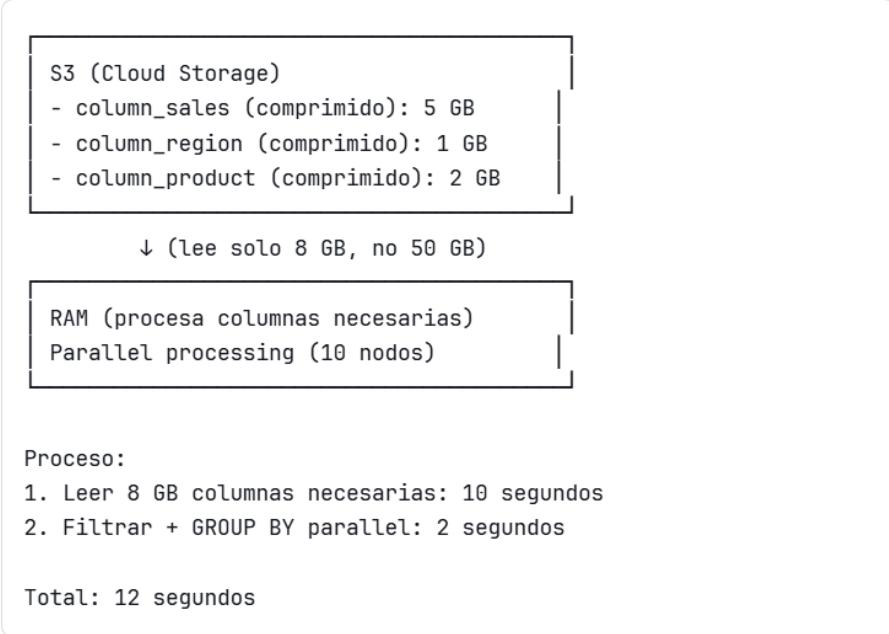
Mismo Query: "SELECT SUM (sales) WHERE region = 'USA' GROUP BY product

Dataset: 100 millones de filas

#### 2.5.3.4.7.3.1: PostgreSQL (Row-Based, DISCO, ROLAP):



#### 2.5.3.4.7.3.2: Redshift (Column-Based, DISCO con cache, ROLAP)



### 2.5.3.4.7.3.3: SSAS MOLAP (Array-Based, DISCO SSD):

SSD (NVMe ultra rápido)  
sales\_cube.mdb: 15 GB (comprimido)  
+ Índices pre-calculados

↓ (Lookup directo, no scan)

RAM (cache de agregaciones)  
Agregación [USA][Products] ya guardada

Proceso:

1. Lookup índice: 0.01 segundos
2. Leer agregación pre-calculada: 0.5 segundos

Total: 0.51 segundos

### 2.5.3.4.7.3.4: SAP HANA (Column-Based, IN-MEMORY):

RAM (512 GB)  
- column\_sales: 5 GB (en RAM)  
- column\_region: 1 GB (en RAM)  
- column\_product: 2 GB (en RAM)  
TODO ya cargado en RAM desde mañana

↓ (0 acceso a disco)

Procesa directamente en RAM (parallel)

Proceso:

1. Leer desde RAM: 0.05 segundos
2. Filtrar + GROUP BY en RAM: 0.02 segundos

Total: 0.07 segundos

### 2.5.3.4.7.3.5: Power BI (Column-Based, IN-MEMORY):

RAM (64 GB)  
Modelo completo cargado:  
- Sales: 3 GB (comprimido VertiPaq)  
- Products: 100 MB  
- Regions: 50 MB  
TODO en RAM desde que abriste .pbix

↓ (0 acceso a disco)

Query engine DAX procesa en RAM

Proceso:

1. Filtrar region = 'USA' en RAM: 0.01 seg
2. SUM + GROUP BY en RAM: 0.02 seg

Total: 0.03 segundos

### 2.5.3.4.7.3.6: HOLAP (Híbrido):

Si query es agregado común:

RAM: Agregaciones  
[USA][Products] = valores pre-calculados

Total: 0.1 segundos (lee RAM)

Si query es detalle específico:

DISCO: Detalles  
Scan columnar de transacciones

Total: 2-5 segundos (lee disco)

## 2.5.3.4.7.4: ⚡ Resumen: ¿Qué usar según necesidades?

### Presupuesto Bajo + Datos Grandes:

- ROLAP en DISCO (PostgreSQL, MySQL)
  - Costo: \$100 para 1TB
  - Velocidad: 10-60 segundos
  - Datos: Ilimitados

### Balance Costo/Velocidad:

- ROLAP Columnar DISCO (Redshift, BigQuery)
  - Costo: \$1,000 para 1TB/mes
  - Velocidad: 1-10 segundos
  - Datos: Ilimitados (cloud)

### Velocidad Crítica + Presupuesto Alto:

- IN-MEMORY (SAP HANA, SSAS Tabular, Power BI)
  - Costo: \$5,000-10,000 para 1TB
  - Velocidad: 0.01-0.1 segundos
  - Datos: Limitado a RAM disponible

### Máxima Velocidad en Dashboards:

- MOLAP DISCO (SSAS MOLAP tradicional)
  - Costo: \$2,000 para cubo 500GB
  - Velocidad: 0.1-0.5 segundos
  - Datos: Limitado (explosión dimensional)

### Mejor Balance General:

- HOLAP (Agregaciones RAM + Detalles DISCO)
  - Costo: \$2,000-3,000
  - Velocidad: 0.1-2 segundos (depende query)
  - Datos: Flexible

## 2.5.3.4.7.5: Resumen Visual: Row vs Column vs Array Storage

Mismos 6 registros en 3 formatos diferentes:

Datos originales (lógicos):

1. Alice compró Laptop por \$1000 en Enero
2. Alice compró Mouse por \$20 en Enero
3. Bob compró Laptop por \$1000 en Enero
4. Alice compró Mouse por \$20 en Febrero
5. Bob compró Keyboard por \$50 en Febrero
6. Alice compró Laptop por \$1000 en Marzo

### 2.5.3.4.7.5.1: ROW-BASED Storage (PostgreSQL, MySQL)

Archivo: fact\_sales.dat



|                                                   |                   |
|---------------------------------------------------|-------------------|
| [id:1, prod:Laptop, cust:Alice, amt:1000, mo:Jan] | ← Fila 1 completa |
| [id:2, prod:Mouse, cust:Alice, amt:20, mo:Jan]    | ← Fila 2 completa |
| [id:3, prod:Laptop, cust:Bob, amt:1000, mo:Jan]   | ← Fila 3 completa |
| [id:4, prod:Mouse, cust:Alice, amt:20, mo:Feb]    | ← Fila 4 completa |
| [id:5, prod:Keyboard, cust:Bob, amt:50, mo:Feb]   | ← Fila 5 completa |
| [id:6, prod:Laptop, cust:Alice, amt:1000, mo:Mar] | ← Fila 6 completa |

Organización: Cada FILA es una unidad continua en disco

Query "SUM(amount)":

- ✗ Debe leer TODAS las 6 filas completas
- ✗ Desperdicia lectura de id, prod, cust, mo
- ✓ Solo usa column 'amount'

Eficiencia: ~20%

## 2.5.3.4.7.5.2: COLUMN-BASED Storage (Redshift, BigQuery, Snowflake)

Directorio: fact\_sales/



— column\_id.dat

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|

← Todos los IDs juntos

— column\_product.dat

|        |       |        |       |          |        |
|--------|-------|--------|-------|----------|--------|
| Laptop | Mouse | Laptop | Mouse | Keyboard | Laptop |
|--------|-------|--------|-------|----------|--------|

← Todos los productos juntos



— column\_customer.dat

|       |       |     |       |     |       |
|-------|-------|-----|-------|-----|-------|
| Alice | Alice | Bob | Alice | Bob | Alice |
|-------|-------|-----|-------|-----|-------|

← Todos los clientes juntos



— column\_amount.dat ← ↳ SOLO ESTE para SUM

|      |    |      |    |    |      |
|------|----|------|----|----|------|
| 1000 | 20 | 1000 | 20 | 50 | 1000 |
|------|----|------|----|----|------|

← Todos los montos juntos



— column\_month.dat

|     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|
| Jan | Jan | Jan | Feb | Feb | Mar |
|-----|-----|-----|-----|-----|-----|

← Todos los meses juntos

Organización: Cada COLUMNA es una unidad continua en disco

Query "SUM(amount)":

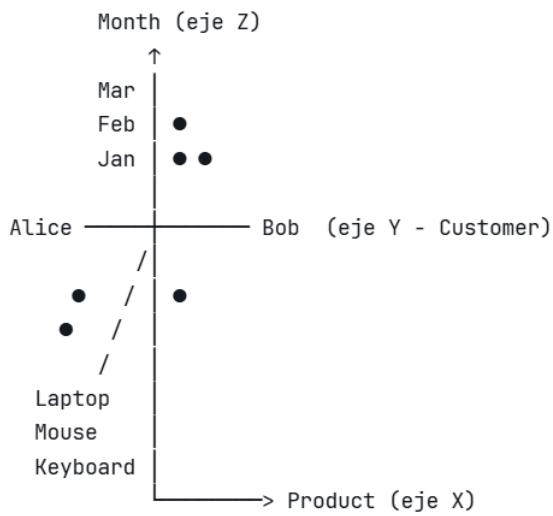
- Solo lee column\_amount.dat
- Ignora completamente otros archivos
- Lee exactamente lo necesario: [1000, 20, 1000, 20, 50, 1000]

Eficiencia: 100%

## 2.5.3.4.7.5.3: ARRAY-BASED Storage (SSAS MOLAP, IBM Cognos)

Archivo: sales\_cube.mdb

Estructura conceptual 3D:



Cada ● = 1 celda con valor pre-calculado

Almacenamiento físico:

Array[product][customer][month] = value

```
[0][0][0] = 1000 ← [Laptop][Alice][Jan]
[1][0][0] = 20 ← [Mouse][Alice][Jan]
[2][0][0] = 0 ← [Keyboard][Alice][Jan]
[0][1][0] = 1000 ← [Laptop][Bob][Jan]
[1][1][0] = 0 ← [Mouse][Bob][Jan]
[2][1][0] = 0 ← [Keyboard][Bob][Jan]
[0][0][1] = 0 ← [Laptop][Alice][Feb]
[1][0][1] = 20 ← [Mouse][Alice][Feb]
[2][0][1] = 0 ← [Keyboard][Alice][Feb]
[0][1][1] = 0 ← [Laptop][Bob][Feb]
[1][1][1] = 0 ← [Mouse][Bob][Feb]
[2][1][1] = 50 ← [Keyboard][Bob][Feb]
[0][0][2] = 1000 ← [Laptop][Alice][Mar]
[1][0][2] = 0 ← [Mouse][Alice][Mar]
[2][0][2] = 0 ← [Keyboard][Alice][Mar]
[0][1][2] = 0 ← [Laptop][Bob][Mar]
[1][1][2] = 0 ← [Mouse][Bob][Mar]
[2][1][2] = 0 ← [Keyboard][Bob][Mar]
```

---

Organización: COORDENADAS en espacio N-dimensional  
NO hay concepto de "fila" ni "columna"

Query "Total Alice":

- Lookup directo: Array['ALL'][Alice]['ALL']
- Lee 1 valor: 2040
- NO calcula nada (ya está guardado)

Eficiencia: Máxima (pre-calculado)

Tiempo: 0.01 segundos

## 2.5.3.4.7.5.4: 🎨 Visualización Física en Disco

### Row-Based (1 archivo):

```
fact_sales.dat (disco duro):
```

```
Sector 1: [Fila 1: todos los campos juntos]
Sector 2: [Fila 2: todos los campos juntos]
Sector 3: [Fila 3: todos los campos juntos]
Sector 4: [Fila 4: todos los campos juntos]
Sector 5: [Fila 5: todos los campos juntos]
Sector 6: [Fila 6: todos los campos juntos]
```

### Column-Based (6 archivos separados):

```
fact_sales/ (directorio en disco):
```

```
column_id.dat: [1,2,3,4,5,6]
column_product.dat: [L,M,L,M,K,L]
column_customer.dat: [A,A,B,A,B,A]
column_amount.dat: [1000,20,1000,20,50,1000] ← Solo este para SUM
column_month.dat: [Jan,Jan,Jan,Feb,Feb,Mar] |
```

### Array-Based (1 archivo especializado):

```
sales_cube.mdb (archivo binario):
```

```
Header: Metadata del cubo
Index: [Laptop→0, Mouse→1, Keyboard→2]
 [Alice→0, Bob→1]
 [Jan→0, Feb→1, Mar→2]
Data: Array[3][2][3] = 18 celdas base
 + Agregaciones = 54 celdas total
Cada celda: coordenadas [x][y][z] → valor
```

## 2.5.3.4.7.5.5: Comparación de Lectura

Query: "¿Cuánto vendió Alice en total?"

### Row-Based:

1. Abre fact\_sales.dat
2. Lee fila 1: [1, Laptop, Alice, 1000, Jan] → es Alice, suma +1000
3. Lee fila 2: [2, Mouse, Alice, 20, Jan] → es Alice, suma +20
4. Lee fila 3: [3, Laptop, Bob, 1000, Jan] → NO es Alice, skip
5. Lee fila 4: [4, Mouse, Alice, 20, Feb] → es Alice, suma +20
6. Lee fila 5: [5, Keyboard, Bob, 50, Feb] → NO es Alice, skip
7. Lee fila 6: [6, Laptop, Alice, 1000, Mar] → es Alice, suma +1000
8. Total:  $1000+20+20+1000 = 2040$

Bytes leídos: 600 bytes (6 filas × 100 bytes)

Bytes usados: ~60 bytes (solo customer+amount de filas relevantes)

Tiempo: 5-30 segundos (con millones de filas)

### Column-Based:

Column Based.

1. Lee column\_customer.dat: [Alice, Alice, Bob, Alice, Bob, Alice]
2. Identifica índices Alice: [0, 1, 3, 5]
3. Lee column\_amount.dat solo en esos índices: [1000, 20, 20, 1000]
4. Suma:  $1000+20+20+1000 = 2040$

Bytes leídos: ~80 bytes (column\_customer + 4 valores amount)

Bytes usados: ~80 bytes (casi todo)

Tiempo: 1-5 segundos (scan columnar optimizado)

### Array-Based:

1. Lookup en índice: Alice → 0
2. Accede a agregación: AggArray['ALL'][0]['ALL']
3. Lee valor: 2040
4. FIN (no calcula nada, valor ya guardado)

Bytes leídos: 8 bytes (1 valor pre-calculado)

Bytes usados: 8 bytes (todo)

Tiempo: 0.01-0.1 segundos (acceso directo)

#### 2.5.3.4.7.5.6: 📈 Resumen Final: ¿Cuál usar?

| Necesitas                       | Usa          | Razón                                             |
|---------------------------------|--------------|---------------------------------------------------|
| Transacciones rápidas (OLTP)    | Row-Based    | Lee/escribe filas completas rápido                |
| Análisis ad-hoc (SQL flexible)  | Column-Based | Lee solo columnas necesarias                      |
| Dashboards drill-down (BI fijo) | Array-Based  | Valores pre-calculados ultra rápidos              |
| Balance perfecto                | HOLAP        | Combina Array (agregados) + Column/Row (detalles) |

#### 2.5.3.4.7.5.7: 🌐 Arquitectura Real Moderna (2024):

Core DWH:

- PostgreSQL (Row-based) para OLTP
- ó Redshift/BigQuery (Column-based) para OLAP



Data Marts:

- SSAS Tabular (Column-based in-memory) ← Más común hoy
- SSAS Multidimensional (Array-based MOLAP) ← Legacy/casos específicos
- ó Power BI (Column-based in-memory VertiPaq) ← Tendencia actual

Resultado: Column-based ganando terreno, Array-based menos usado

## 2.5.3.4.7.6: Visualización Física: Cómo se Almacenan REALMENTE

### 2.5.3.4.7.6.1: ROLAP – Archivos en Sistemas de Archivos

```
/var/lib/postgresql/data/
└── base/
 └── 16384/ (database ID)
 ├── 16385 (fact_sales table)
 │ └── [Row 1][Row 2][Row 3]...[Row 6]
 │ Cada row: 100 bytes
 ├── 16386 (dim_products table)
 │ └── [Row 1][Row 2][Row 3]
 ├── 16387 (dim_customers table)
 │ └── [Row 1][Row 2]
 └── 16388 (dim_cities table)
 └── [Row 1][Row 2]
```

Lectura: Secuencial (lento)

Query: Escanea filas una por una

### 2.5.3.4.7.6.2: MOLAP – Archivo Binario Especializado

```
/opt/ssas/cubes/
└── sales_cube.mdb
 ├── [HEADER: 1KB]
 │ ├── Dimension definitions
 │ ├── Hierarchy structures
 │ └── Index pointers
 ├── [DIMENSION INDEX: 2KB]
 │ ├── Products: {Laptop→0, Mouse→1, Keyboard→2}
 │ ├── Customers: {Alice→0, Bob→1}
 │ └── Time: {Jan→0, Feb→1, Mar→2}
 └── [DATA ARRAY: 3KB]
 └── Array[producto][cliente][mes]
 ├── [0][0][0] = 1000 ← Laptop-Alice-Jan
 ├── [0][0][1] = 0 ← Laptop-Alice-Feb
 ├── [0][0][2] = 1000 ← Laptop-Alice-Mar
 ├── [0][1][0] = 1000 ← Laptop-Bob-Jan
 ├── ...
 └── [2][1][2] = 0 ← Keyboard-Bob-Mar
```

Lectura: Acceso directo (rápido)

Query: Va directo a posición [x][y][z]

## 2.5.3.4.7.6.3: HOLAP - Dos Almacenamientos Simultáneos: IMPORTANTE

Sistema híbrido:

```
/opt/holap/
└── aggregations/ (MOLAP)
 └── sales_agg.mdb
 └── Array solo con totales/subtotales
 Tamaño: 500 bytes (10% del total)

└── details/ (ROLAP)
 └── /var/lib/postgresql/data/fact_sales.dat
 └── Tablas relacionales con detalles
 Tamaño: 600 bytes (resto)
```

Query Engine decide dinámicamente:

- ¿Pregunta agregada? → Lee MOLAP
- ¿Pregunta detallada? → Lee ROLAP

## 2.5.3.4.7.7: Comparación Visual de Almacenamiento

Mismo dato: "Alice compró Laptop por \$1,000 en Enero"

### ROLAP:

```
fact_sales.dat:
[1, 101, 201, 301, 1000, '2024-01-15']
 ↓ ↓
dim_products.dat: dim_customers.dat:
[101, 'Laptop', ...] [201, 'Alice', ...]

Dato guardado 1 VEZ (normalizado)
Agregación: Se calcula cuando lo pides
```

### MOLAP:

```
sales_cube.mdb:
[Laptop][Alice][Jan] = 1000 ← Nivel detalle
[Laptop][Alice][ALL] = 2000 ← Agregado por Alice
[ALL][Alice][Jan] = 1020 ← Agregado por Enero
[ALL][Alice][ALL] = 2040 ← Total Alice
[ALL][ALL][ALL] = 3090 ← Gran total
```

Dato guardado 5+ VECES (en diferentes agregaciones)  
Agregación: Ya está guardada

### HOLAP:

```
fact_sales.dat (detalle):
[1, 101, 201, 301, 1000, '2024-01-15']

sales_agg.mdb (solo agregaciones):
[Laptop][Alice][ALL] = 2000
[ALL][Alice][ALL] = 2040

Dato guardado 1 VEZ en detalle
Agregaciones guardadas aparte
```

### 2.5.3.4.7.7.1: ⚙ Tabla Comparativa: Almacenamiento Físico

| Aspecto         | ROLAP              | MOLAP                  | HOLAP        |
|-----------------|--------------------|------------------------|--------------|
| Estructura      | Tablas (rows/cols) | Arrays N-dimensionales | Ambos        |
| Archivo         | .dat (PostgreSQL)  | .mdb (cubo binario)    | .dat + .mdb  |
| Agregaciones    | ✗ NO guardadas     | ✓ TODAS guardadas      | ⚠ Selectivas |
| Redundancia     | Baja (normalizado) | Alta (pre-calculado)   | Media        |
| Tamaño ejemplo  | 980 bytes          | 2,500 bytes            | 1,100 bytes  |
| Lectura         | Secuencial (scan)  | Directo (index)        | Inteligente  |
| Velocidad query | 5-30 seg           | 0.01-0.1 seg           | 0.1-2 seg    |
| Espacio disco   | ● Bajo             | ● Alto                 | ● Medio      |
| Flexibilidad    | ✓ Alta             | ✗ Baja                 | ⚠ Media      |
| Build time      | ✓ No requiere      | ✗ Horas                | ⚠ Minutos    |

## 2.5.3.4.7.8: Ejemplo Real: E-commerce con 1M transacciones

### ROLAP:

```
fact_sales.dat: 1,000,000 filas × 100 bytes = 100 MB
dim_products.dat: 10,000 productos × 500 bytes = 5 MB
dim_customers.dat: 100,000 clientes × 200 bytes = 20 MB
dim_time.dat: 365 días × 100 bytes = 36 KB
```



TOTAL: ~125 MB

### MOLAP:

Dimensiones:

- Products: 10,000
- Customers: 100,000
- Time: 365 días
- Cities: 1,000

Celdas base teóricas:  $10K \times 100K \times 365 \times 1K = 365$  trillones

Celdas reales (sparsity 99%): ~365 millones

Con agregaciones: ~1 mil millones de celdas

Tamaño:  $1,000,000,000 \times 8$  bytes = 8 GB (sin comprimir)

Tamaño comprimido: ~2-3 GB

TOTAL: ~2.5 GB

### HOLAP:

```
Details (ROLAP): 125 MB (datos raw)
```



```
Aggregations (MOLAP): 500 MB (solo agregaciones comunes)
```

TOTAL: ~625 MB

## 2.5.3.4.7.8.1: 📈 Cuándo Usar Cada Uno

### Usar ROLAP si:

- Necesitas flexibilidad total
- Queries ad-hoc impredecibles
- Presupuesto de disco limitado
- Datos cambian frecuentemente
- Muchas dimensiones (>10)
- No necesitas queries <1 segundo

### Usar MOLAP si:

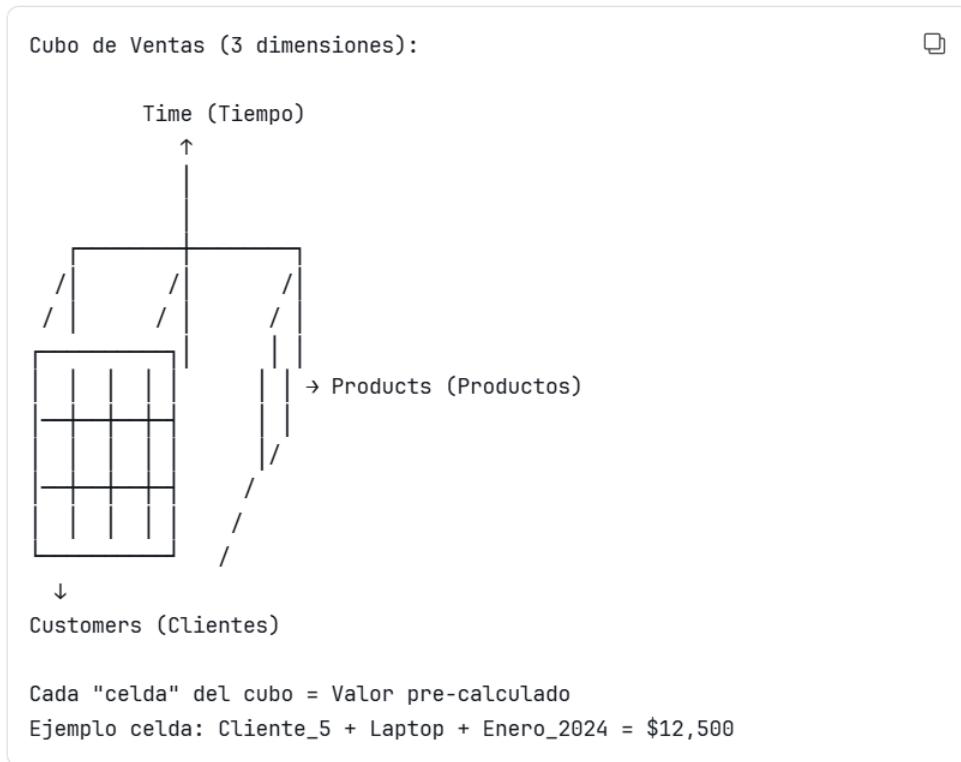
- Necesitas queries <0.1 segundos
- Queries predecibles (dashboards fijos)
- Presupuesto de disco amplio
- Datos estables (actualización diaria/semanal)
- Pocas dimensiones (<10)
- Drill-down frecuente

### Usar HOLAP si:

- Necesitas balance
- Queries comunes rápidas + Queries raras flexibles
- Presupuesto moderado
- Mezcla de análisis agregado y detallado
- Mejor opción para mayoría de casos reales

## 2.5.3.5: 5 Anatomía Detallada de un OLAP Cube

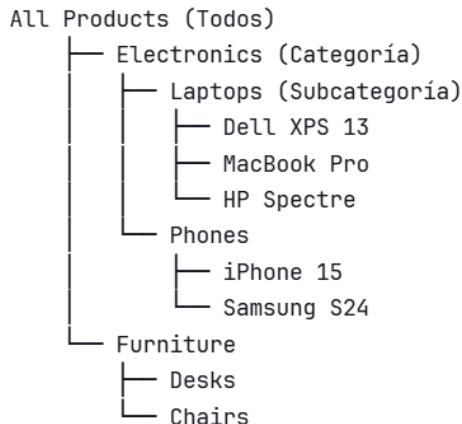
### Concepto Visual: El Cubo



## 2.5.3.5.1: Las 3 Dimensiones Explicadas

### 2.5.3.5.1.1: Dimensión 1: Products (Productos)

Jerarquía de Productos:

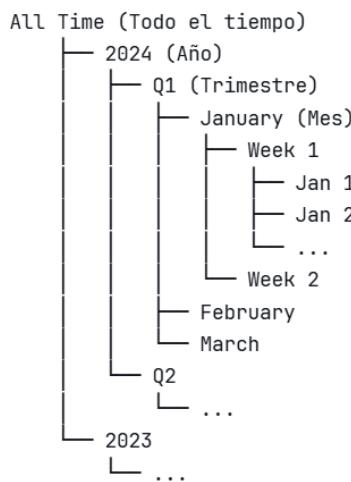


Niveles:

1. All Products (nivel más alto)
2. Category (categoría)
3. Subcategory (subcategoría)
4. Product (producto específico - nivel más bajo)

### 2.5.3.5.1.2: Dimensión 2: Time (Tiempo)

Jerarquía de Tiempo:

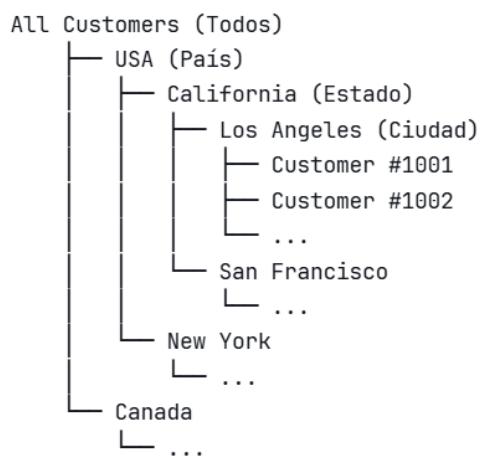


Niveles:

1. All Time
2. Year (año)
3. Quarter (trimestre)
4. Month (mes)
5. Week (semana)
6. Day (día - nivel más bajo)

### 2.5.3.5.1.3: Dimensión 3: Customers (Clientes)

Jerarquía de Clientes:



Niveles:

1. All Customers
2. Country (país)
3. State (estado)
4. City (ciudad)
5. Customer (cliente específico - nivel más bajo)

## 2.5.3.5.2: 12 Pre-Calculación: El Secreto del OLAP Cube

¿Qué significa "Pre-calculado"?

Cuando el cubo se CONSTRUYE, calcula TODAS las combinaciones posibles y las GUARDA.

Ejemplo Detallado: Cubo de Ventas

Setup:

Dimensiones:

- Products: 3 productos (Laptop, Mouse, Keyboard)
- Time: 3 meses (Jan, Feb, Mar)
- Customers: 2 clientes (Alice, Bob)

Total combinaciones:  $3 \times 3 \times 2 = 18$  celdas base

### 2.5.3.5.2.1: Paso 1: Datos Raw (Sin procesar)

Tabla transaccional (fact\_sales):

| sale_id | product  | month | customer | amount |
|---------|----------|-------|----------|--------|
| 1       | Laptop   | Jan   | Alice    | 1000   |
| 2       | Mouse    | Jan   | Alice    | 20     |
| 3       | Laptop   | Jan   | Bob      | 1000   |
| 4       | Mouse    | Feb   | Alice    | 20     |
| 5       | Keyboard | Feb   | Bob      | 50     |
| 6       | Laptop   | Mar   | Alice    | 1000   |

Total filas: 6

### 2.5.3.5.2.2: Paso 2: Construcción del Cubo (Precálculo)

El cubo calcula y guarda TODAS las agregaciones:

## Nivel 1: Combinaciones específicas (18 celdas base)

[Laptop, Jan, Alice] = \$1,000

[Mouse, Jan, Alice] = \$20

[Keyboard, Jan, Alice] = \$0

[Laptop, Jan, Bob] = \$1,000

[Mouse, Jan, Bob] = \$0

[Keyboard, Jan, Bob] = \$0

[Laptop, Feb, Alice] = \$0

[Mouse, Feb, Alice] = \$20

[Keyboard, Feb, Alice] = \$0

[Laptop, Feb, Bob] = \$0

[Mouse, Feb, Bob] = \$0

[Keyboard, Feb, Bob] = \$50

[Laptop, Mar, Alice] = \$1,000

[Mouse, Mar, Alice] = \$0

[Keyboard, Mar, Alice] = \$0

[Laptop, Mar, Bob] = \$0

[Mouse, Mar, Bob] = \$0

[Keyboard, Mar, Bob] = \$0

## Nivel 2: Agregaciones por 2 dimensiones

## Recordatorio de setup:

### Setup:

Dimensiones:

- Products: 3 productos (Laptop, Mouse, Keyboard)
- Time: 3 meses (Jan, Feb, Mar)
- Customers: 2 clientes (Alice, Bob)

Total combinaciones:  $3 \times 3 \times 2 = 18$  celdas base

## Por Producto + Mes (ignora Customer):

[Laptop, Jan] = \$1,000 + \$1,000 = \$2,000

[Mouse, Jan] = \$20

[Keyboard, Jan] = \$0

[Laptop, Feb] = \$0

[Mouse, Feb] = \$20

[Keyboard, Feb] = \$50

[Laptop, Mar] = \$1,000

[Mouse, Mar] = \$0

[Keyboard, Mar] = \$0

## Por Producto + Customer (ignora Time)

[Laptop, Alice] = \$1,000 + \$1,000 = \$2,000

[Mouse, Alice] = \$20 + \$20 = \$40

[Keyboard, Alice] = \$0

[Laptop, Bob] = \$1,000

[Mouse, Bob] = \$0

[Keyboard, Bob] = \$50

## Por Mes + Customer (ignora Product)

[Jan, Alice] = \$1,000 + \$20 = \$1,020

[Feb, Alice] = \$20

[Mar, Alice] = \$1,000

[Jan, Bob] = \$1,000

[Feb, Bob] = \$50

[Mar, Bob] = \$0

## Nivel 3: Agregación por 1 dimensión

**Por producto (ignora Time y Customer):**

---

[Laptop] = \$1,000 + \$1,000 + \$1,000 = \$3,000

[Mouse] = \$20 + \$20 = \$40

[Keyboard] = \$50

**Por Mes (ignora Product y Customer):**

---

[Jan] = \$1,000 + \$20 + \$1,000 = \$2,020

[Feb] = \$20 + \$50 = \$70

[Mar] = \$1,000

**Por Customer (Ignora Product y Time):**

---

[Alice] = \$1,000 + \$20 + \$20 + \$1,000 = \$2,040

[Bob] = \$1,000 + \$50 = \$1,050

**Nivel 4: Total General**

[ALL] = \$1,000 + \$20 + \$1,000 + \$20 + \$50 + \$1,000 = \$3,090

### **2.5.3.5.2.3: Resumen de Pre-Cálculos**

Celdas base (específicas): 18  
Agregaciones 2D: 27  
Agregaciones 1D: 8  
Total general: 1

TOTAL CELDAS EN EL CUBO: 54 valores pre-calculados

Con solo 6 filas de datos raw → 54 valores guardados

#### **[transcripción]**

Celdas base (específicas): 18  
Agregaciones 2D: 27  
Agregaciones 1D: 8  
Total general: 1  
TOTAL CELDAS EN CUBO: 54 valores pre-calculados  
Con solo 6 filas de datos raw -> 54 (18 filas x 3 columnas) valores guardados

### **Esto crece EXPONENCIALMENTE:**

3 dimensiones con 10 valores cada una:



Celdas base:  $10 \times 10 \times 10 = 1,000$

Total celdas con agregaciones: ~2,000

10 dimensiones con 10 valores cada una:

Celdas base:  $10^{10} = 10,000,000,000$  (10 mil millones)

Total celdas: ~20 mil millones

#### **2.5.3.5.2.4: Explicación de los cálculos de [Resumen de Pre-Calculos]**

Tenemos: - 3 productos - 3 meses - 2 clientes

---

#### **Celdas base: 18**

---

Son TODAS las combinaciones posibles:

3 productos x 3 meses x 2 clientes = 18 celdas

[Laptop][Jan][Alice]  
[Laptop][Jan][Bob]  
[Laptop][Feb][Alice]  
[Laptop][Feb][Bob]  
... (18 en total)

---

#### **Agregaciones: 2D: 27**

---

**Agrupamos por 2 dimensiones (ignoramos 1)**

**Grupo 1: [Producto x Mes] (ignora cliente)**

3 productos x 3 meses = 9 agragaciones

[Laptop][Jan] = suma de Alice + Bob  
[Laptop][Feb] = suma de Alice + Bob  
[Laptop][Mar] = suma de Alice + Bob  
[Mouse][Jan] = ...  
[Mouse][Feb] = ...  
[Mouse][Mar] = ...  
[Keyboard][Jan] = ...  
[Keyboard][Feb] = ...  
[Keyboard][Mar] = ...

**Grupo 2: [Producto x Cliente] (ignora mes)**

3 productos x 2 clientes = 6 agragaciones

[Laptop][Alice] = suma de todos los meses  
[Laptop][Bob] = suma de todos los meses  
[Mouse][Alice] = ...  
[Mouse][Bob] = ...  
[Keyboard][Alice] = ...  
[Keyboard][Bob] = ...

**Grupo 3: [Mes x Cliente] (ignora producto)**

3 meses x 2 clientes = 6 agragaciones

[Jan][Alice] = suma de todos los productos  
[Jan][Bob] = ...  
[Feb][Alice] = ...  
[Feb][Bob] = ...  
[Mar][Alice] = ...  
[Mar][Bob] = ...

**Grupo 4: [Producto solo] (ignora mes y cliente)**

3 productos = 3 agregaciones

[Laptop] = suma total de Laptop

[Mouse] = suma total de Mouse

[Keyboard] = suma total de Keyboard

**Grupo 5: [Mes solo] (ignora producto y cliente)**

3 meses = 3 agregaciones

[Jan] = suma total de Enero

[Feb] = suma total de Febrero

[Mar] = suma total de Marzo

**Grupo 6: [Cliente solo] (ignora producto y mes)**

2 clientes = 2 agregaciones

[Alice] = suma total de Alice

[Bob] = suma total de Bob

---

**Total:** 9 (2D) + 6 (2D) + 6 (2D) = 21

---

**Agregaciones 1D: 8**

---

3 productos + 3 meses + 2 clientes = 8

---

**Total general: 1**

---

**[Brand Total]** = suma de TODO

---

## Por Qué Total General = 1

Total general: 1  
└ Es UNA SOLA celda con el gran total

---

---

Ejemplo:

[ALL][ALL][ALL] = \$3,090  
↑  
UNA celda que suma TODO

---

---

Desglose:

|           |                                       |
|-----------|---------------------------------------|
| Laptop:   | \$1,000 + \$1,000 + \$1,000 = \$3,000 |
| Mouse:    | \$20 + \$20 = \$40                    |
| Keyboard: | \$50 = \$50                           |
|           | _____                                 |
|           | TOTAL: \$3,090                        |

Esto se guarda en 1 SOLA celda: [ALL] = \$3,090

---

**Clave:** Es **1 valor** (no 1 fila), que representa la suma de absolutamente todo en el cubo.

Query que la usa:



```
SELECT SUM(amount) FROM sales;
-- No WHERE, no GROUP BY
-- Solo el gran total
-- Resultado instantáneo: lee 1 celda pre-calculada
```

Por eso "Total general: 1" = una celda con el valor agregado máximo.

### 2.5.3.5.3: ⚡ Velocidad: ROLAP vs MOLAP

Escenario: "¿Ventas totales de Laptops en Enero?"

ROLAP (Base de datos relacional):



```
-- Query ejecuta en tiempo real
SELECT SUM(amount)
FROM fact_sales
WHERE product = 'Laptop'
AND month = 'January';
```

Proceso:

1. Escanear tabla fact\_sales (millones de filas)
2. Filtrar por product = 'Laptop'
3. Filtrar por month = 'January'
4. Sumar amounts
5. Devolver resultado

Tiempo: 5-30 segundos (dependiendo tamaño datos)

MOLAP (OLAP Cube):



```
-- Lookup directo en array pre-calculado
cube[Laptop][January][ALL_CUSTOMERS]
```

Proceso:

1. Buscar posición en array: [índice\_Laptop][índice\_January]  
[índice\_ALL]
2. Leer valor guardado
3. Devolver resultado

Tiempo: 0.01-0.1 segundos (solo lectura de memoria)

Diferencia: 50-300x más rápido

### 2.5.3.5.4: 🔎 Drill-Down, Slice & Dice - Operaciones Interactivas

#### 2.5.3.5.4.1: Drill-Down (Profundizar)

Qué es: Navegar de nivel GENERAL a nivel ESPECÍFICO en una jerarquía

## Ejemplo Visual:

```
● ● ●
Usuario en dashboard ve:

Nivel 1: Total ventas = $3,090
↓ [Click en "Ver por producto"]

Nivel 2:
└─ Laptop: $3,000
└─ Mouse: $40
└─ Keyboard: $50
 ↓ [Click en "Laptop"]

Nivel 3: Laptop por mes
└─ January: $2,000
└─ February: $0
└─ March: $1,000
 ↓ [Click en "January"]

Nivel 4: Laptop en January por cliente
└─ Alice: $1,000
└─ Bob: $1,000
```

## Por qué es rápido en OLAP Cube:

Cada click = Leer posición diferente del array (ya calculado)  
NO requiere nueva query a la base de datos

### 2.5.3.5.4.2: Slice (Rebanar)

Qué es: Cortar el cubo en una dimensión específica, fijando un valor

#### Ejemplo Visual:

Cubo completo (3D):  
Products × Time × Customers



Slice por "January" (fijo):  
Products × Customers (2D)  
(solo datos de Enero)

Resultado visual:

|          | Alice   | Bob     |
|----------|---------|---------|
| Laptop   | \$1,000 | \$1,000 |
| Mouse    | \$20    | \$0     |
| Keyboard | \$0     | \$0     |

Solo muestra datos de ENERO (dimensión Time fijada)

### 2.5.3.5.4.3: Dice (Cortar en cubos)

Qué es: Seleccionar sub-cubo con rangos específicos en múltiples dimensiones.

#### Ejemplo Visual:

Cubo completo:

Products: [Laptop, Mouse, Keyboard]

Time: [Jan, Feb, Mar]

Customers: [Alice, Bob]



Dice selección:

Products: [Laptop, Mouse] (solo 2)

Time: [Jan, Feb] (solo 2)

Customers: [Alice] (solo 1)

Sub-cubo resultante (2×2×1):

January February

|        |         |      |       |
|--------|---------|------|-------|
| Laptop | \$1,000 | \$0  | Alice |
| Mouse  | \$20    | \$20 | Alice |

## 2.5.3.5.5: MDX - Lenguaje de OLAP Cubes + Ejemplo MDX Detallado

### MDX vs SQL

SQL (para ROLAP):

```
SELECT
 product_name,
 SUM(amount) as total_sales
FROM fact_sales
WHERE month = 'January'
GROUP BY product_name;
```

MDX (para MOLAP):

```
SELECT
 [Products].[Product].Members ON ROWS,
 [Measures].[Sales] ON COLUMNS
FROM [Sales Cube]
WHERE [Time].[Month].[January]
```

## Ejemplo MDX Detallado

Pregunta: "Ventas de todos los productos en Q1 2024 para clientes en California"

mdx



```
SELECT
 [Products].[Category].Children ON ROWS,
 [Time].[Quarter].[Q1 2024] ON COLUMNS
FROM [Sales Cube]
WHERE [Customers].[State].[California]
```

Resultado:

|             | Q1 2024  |
|-------------|----------|
| Electronics | \$50,000 |
| Furniture   | \$20,000 |

## Explicación:

```
[Products].[Category].Children
└─ Obtener todas las categorías (Electronics, Furniture, etc) ✓

[Time].[Quarter].[Q1 2024]
└─ Fijar tiempo en Q1 2024

WHERE [Customers].[State].[California]
└─ Solo clientes de California
```

## 2.5.3.6: Bases de Datos Multidimensionales

### ¿Qué son?

Hardware/Software especializado para almacenar OLAP Cubes

Diferencia con BD Relacionales:

BD Relacional (ROLAP):

- Almacenamiento: Tablas en disco
- Estructura: Filas y columnas
- Query: SQL en tiempo real
- Tecnología: PostgreSQL, MySQL, Oracle



BD Multidimensional (MOLAP):

- Almacenamiento: Arrays multidimensionales
- Estructura: Dimensiones y celdas
- Query: MDX (lectura pre-calculada)
- Tecnología: Microsoft SSAS (MOLAP), Oracle OLAP, IBM Cognos

### 2.5.3.6.1: Ejemplo: Microsoft SSAS (SQL Server Analysis Services)

Como se almacena físicamente:

Archivo del cubo en disco:

/data/sales\_cube.mdb

Contenido (simplificado):

Metadata:  
- Dimension: Products (1000 items)  
- Dimension: Time (365 days)  
- Dimension: Customers (50,000)  
Total cells: 18,250,000,000

Pre-calculated data:  
[0][0][0] = \$1,234.56  
[0][0][1] = \$234.00  
[0][0][2] = \$5,678.90  
... (18 mil millones de valores)

Tamaño archivo: 200-500GB (comprimido)

## 2.5.3.6.2: Tamaño de OLAP Cubes - Explosión Exponencial

### Problema: Curse of Dimensionality

Ejemplo realista:



Dimensiones:

- Products: 10,000 productos
- Time: 365 días × 3 años = 1,095 días
- Customers: 100,000 clientes
- Stores: 500 tiendas
- Sales Channels: 10 canales (web, retail, phone, etc)

Celdas base:

$10,000 \times 1,095 \times 100,000 \times 500 \times 10 = 5,475,000,000,000,000$  celdas  
(5.5 cuatrillones de celdas)

Con agregaciones: ~10 cuatrillones de celdas

Tamaño estimado: 100+ TB

### Solución:

1. Sparsity (escasez): No guardar celdas vacías

Ejemplo: No todas las tiendas venden todos los productos

Ahorro: 70-90% espacio

2. Particionamiento: Dividir cubo en sub-cubos

Ejemplo: Un cubo por año

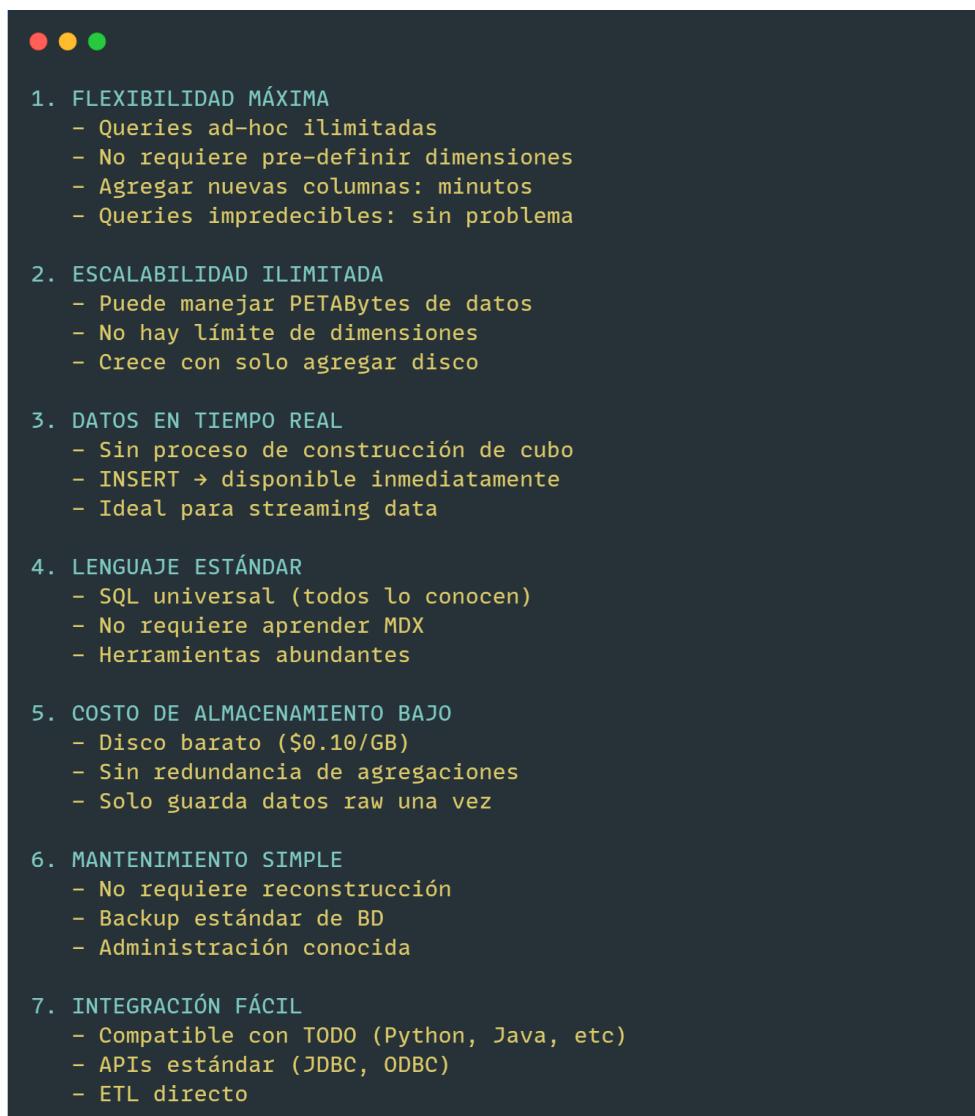
3. Agregaciones parciales: No calcular TODAS las combinaciones

Solo calcular las más usadas

## 2.5.3.6.3: ✓✗ Ventajas y Desventajas: ROLAP vs MOLAP vs HOLAP

### 2.5.3.6.3.1: ● ROLAP (Relational OLAP)

#### ✓ VENTAJAS:



#### ✗ DESVENTAJAS:



1. VELOCIDAD LENTA
  - Calcula agregaciones en tiempo real
  - Queries complejas: 10-60 segundos
  - No apto para dashboards interactivos
2. CARGA EN PRODUCCIÓN
  - Queries pesadas afectan OLTP
  - Requiere separación física (DWH)
  - Bloqueos en hora pico
3. SIN PRE-CÁLCULO
  - Cada query recalcula TODO
  - "SUM(sales) GROUP BY region" siempre lento
  - No aprende de queries pasadas
4. OPTIMIZACIÓN MANUAL
  - Requiere índices manuales
  - Particionamiento manual
  - Tuning constante
5. NO OPTIMIZADO PARA JERARQUÍAS
  - Drill-down requiere múltiples queries
  - "Año → Mes → Día" = 3 queries separadas
  - Experiencia usuario inferior
6. DESPERDICIO DE I/O (Row-based)
  - Lee datos innecesarios
  - "SELECT amount" lee fila completa
  - Eficiencia baja en análisis

## 2.5.3.6.3.2: MOLAP (Multidimensional OLAP)

### ✓ VENTAJAS:



#### 1. VELOCIDAD EXTREMA

- Queries: 0.01-0.1 segundos
- Agregaciones pre-calculadas
- Ideal para dashboards tiempo real

#### 2. DRILL-DOWN INSTANTÁNEO

- Año → Mes → Día: 0.1 segundos total
- Jerarquías optimizadas nativamente
- Experiencia usuario excelente

#### 3. SLICE & DICE RÁPIDO

- Cambiar perspectivas: instantáneo
- Rotar dimensiones: sin re-calcular
- Análisis exploratorio fluido

#### 4. OPTIMIZADO PARA BI

- Diseñado específicamente para análisis
- Herramientas BI nativas (Excel, Tableau)
- Experiencia perfecta para usuarios negocio

#### 5. COMPRESIÓN ALTA

- Sparse arrays (celdas vacías no ocupan espacio)
- Datos repetidos comprimidos
- 10:1 a 100:1 ratio común

#### 6. CÁLCULOS COMPLEJOS PRE-CALCULADOS

- Ratios, porcentajes, YoY
- Métricas derivadas guardadas
- No requiere re-calcular

#### 7. CONSISTENCIA DE DATOS

- Todos ven mismos números
- Lógica negocio centralizada
- Sin discrepancias entre reportes

## DESVENTAJAS:

### 1. INFLEXIBILIDAD

- Dimensiones fijas al construir cubo
- Nueva dimensión = reconstruir cubo completo
- No soporta queries ad-hoc

### 2. EXPLOSIÓN EXPONENCIAL

- 10 dimensiones × 1000 valores = trillones celdas
- Crece exponencialmente
- Limitado a 8-10 dimensiones prácticas

### 3. CONSTRUCCIÓN LENTA

- Build inicial: 2-8 horas (o más)
- Full rebuild semanal
- Downtime durante construcción

### 4. COSTO DE ALMACENAMIENTO ALTO

- Guarda TODAS las agregaciones
- 10-100x más espacio que ROLAP
- Datos redundantes

### 5. DATOS NO EN TIEMPO REAL

- Actualización: diaria/semanal típicamente
- Latencia: horas
- No apto para streaming

### 6. LENGUAJE ESPECIALIZADO

- MDX complejo
- Curva aprendizaje alta
- Menos desarrolladores conocen

### 7. MANTENIMIENTO COMPLEJO

- Requiere expertise especializado
- Monitoring de construcción
- Particionamiento manual

### 8. VENDOR LOCK-IN

- Microsoft SSAS domina mercado
- Migrar = reconstruir TODO
- Licencias caras

### 9. NO APTO PARA DETALLE

- Solo agregaciones
- No puedes ver transacciones individuales
- Requiere volver a ROLAP para drill-through

### 10. HARDWARE ESPECIALIZADO

- Servidores dedicados
- RAM abundante necesaria
- No escala horizontalmente fácil

## 2.5.3.6.3.3: 🥑 HOLAP (Hybrid OLAP)

### ✓ VENTAJAS:

- ● ●
- 1. MEJOR DE AMBOS MUNDOS
  - Velocidad de MOLAP para agregaciones
  - Flexibilidad de ROLAP para detalles
  - Balance óptimo
- 2. COSTO OPTIMIZADO
  - Solo guarda agregaciones frecuentes
  - Detalles en disco barato
  - ~50% costo de MOLAP puro
- 3. ESCALABILIDAD FLEXIBLE
  - Agregaciones limitadas (caben en RAM/SSD)
  - Detalles ilimitados (en ROLAP)
  - Crece según necesidad
- 4. DATOS RECIENTES RÁPIDOS
  - Hot data (últimos 3 meses) en MOLAP
  - Cold data (histórico) en ROLAP
  - Acceso inteligente
- 5. DRILL-THROUGH DISPONIBLE
  - Ve agregación rápida
  - Click → Ve detalle individual
  - Sin límites
- 6. CONSTRUCCIÓN MÁS RÁPIDA
  - Solo agrega datos comunes
  - No todas las combinaciones
  - Build: 30 min - 2 hrs (vs 8 hrs MOLAP)
- 7. QUERIES MIXTAS
  - "Total ventas región" → MOLAP (0.1 seg)
  - "Listar transacciones >\$1000" → ROLAP (2 seg)
  - Sistema decide automáticamente
- 8. MENOR RIESGO
  - Si MOLAP falla → ROLAP disponible
  - Redundancia natural
  - Downtime reducido

### ✗ DESVENTAJAS:



## 1. COMPLEJIDAD ARQUITECTURA

- Dos sistemas que mantener
- Sincronización necesaria
- Más puntos de fallo

## 2. CONFIGURACIÓN DIFÍCIL

- ¿Qué agregar en MOLAP?
- ¿Qué dejar en ROLAP?
- Decisiones no obvias

## 3. PERFORMANCE INCONSISTENTE

- Queries agregadas: 0.1 seg
- Queries detalle: 5 seg
- Usuario no sabe qué esperar

## 4. REQUIERE QUERY ENGINE INTELIGENTE

- Debe decidir MOLAP vs ROLAP
- Lógica compleja
- Puede elegir mal

## 5. OVERHEAD DE GESTIÓN

- Monitorear dos sistemas
- Logs en dos lugares
- Debugging complejo

## 6. LICENCIAS DOBLES

- Licencia MOLAP (cara)
- Licencia ROLAP (moderada)
- Costo total alto

## 7. EXPERTISE REQUERIDA

- Necesitas expertos en MOLAP Y ROLAP
- Equipo más grande
- Training costoso

## 8. NO ES MÁXIMO EN NADA

- No es TAN rápido como MOLAP puro
- No es TAN flexible como ROLAP puro
- Compromiso en todo

## 2.5.3.6.3.4: PARENTESIS: Drill-Down vs Drill-Through

### DRILL-DOWN

Ir de agregado → más detalle (dentro del cubo)

Ejemplo:

Ventas totales 2024: \$3M

↓ drill-down por trimestre

Q1: \$800K, Q2: \$750K, Q3: \$900K, Q4: \$550K

↓ drill-down por mes

Q1 → Ene: \$250K, Feb: \$280K, Mar: \$270K

Sigue en MOLAP (agregaciones pre-calculadas)

### DRILL-THROUGH

Ir de agregado -> transacciones individuales (salir del cubo)

Ejemplo:

Ventas Laptop Enero: \$2,000

↓ drill-through

Ver las 3 transacciones:

- Trans #1: Alice, \$1,000, 2024-01-05
- Trans #3: Bob, \$1,000, 2024-01-12
- Trans #7: Carol, \$0, 2024-01-28

Sale del MOLAP → va a ROLAP/tabla raw

### Resumen:

- **Drill-down:** Más detalle, mismo cubo
- **Drill-through:** Salir a datos raw

#### 2.5.3.6.4: Tabla Comparativa Consolidada + Decisión rápida

| Aspecto              | ROLAP                       | MOLAP                         | HOLAP                      |
|----------------------|-----------------------------|-------------------------------|----------------------------|
| Velocidad queries    | ● Lenta (10-60 seg)         | ● Ultra rápida (0.01-0.1 seg) | ● Media-Rápida (0.1-5 seg) |
| Flexibilidad         | ● Total (queries ad-hoc)    | ● Baja (dimensiones fijas)    | ● Media (depende parte)    |
| Escalabilidad datos  | ● Ilimitada (PB)            | ● Limitada (TB)               | ● Alta (TB-PB)             |
| Tiempo real          | ● Sí (instantáneo)          | ● No (latencia horas)         | ● Parcial (agregados no)   |
| Drill-down           | ● Lento (múltiples queries) | ● Instantáneo                 | ● Rápido (agregados)       |
| Costo almacenamiento | ● Bajo (\$0.10/GB)          | ● Alto (\$5-10/GB RAM)        | ● Medio                    |
| Construcción         | ● No requiere               | ● Larga (2-8 hrs)             | ● Media (30min-2hrs)       |
| Mantenimiento        | ● Simple                    | ● Complejo                    | ● Muy complejo             |
| Lenguaje             | ● SQL (universal)           | ● MDX (especializado)         | ● Ambos (MDX+SQL)          |
| Datos detalle        | ● Sí (completo)             | ● No (solo agregados)         | ● Sí (ROLAP parte)         |
| Usuarios negocio     | ● Difícil (SQL técnico)     | ● Fácil (Excel, BI tools)     | ● Media                    |
| Compresión           | ● Baja                      | ● Alta (10-100:1)             | ● Media                    |
| Consistencia         | ● Variable                  | ● Total (pre-calc)            | ● Media                    |
| Dimensiones max      | ● Ilimitadas                | ● 8-10 prácticas              | ● 10-15                    |
| Mejor para           | Data science, ETL           | Dashboards, BI                | Balance general            |

## 2.5.3.6.4.1: ⏲ Decisión Rápida: ¿Cuál elegir?

Elige ROLAP si:

- Necesitas queries ad-hoc impredecibles
- Datos cambian constantemente (tiempo real)
- Muchas dimensiones (>10)
- Datos muy grandes (>5 TB)
- Presupuesto limitado
- Equipo sabe SQL (no MDX)
- Flexibilidad > Velocidad



Ejemplos:

- Data Lake para data scientists
- ETL/transformaciones
- Reportes mensuales (no interactivos)

Elige MOLAP si:

- Dashboards ejecutivos interactivos
- Drill-down frecuente (año → mes → día)
- Pocas dimensiones (<8)
- Queries predecibles (BI fijo)
- Velocidad crítica (<0.1 seg)
- Datos estables (actualización diaria/semanal)
- Usuarios negocio (Excel, Tableau)
- Velocidad > Todo



Ejemplos:

- Dashboard CEO
- Análisis financiero interactivo
- Reportes regulatorios con drill-down

Elige HOLAP si:

- Necesitas balance
- Queries comunes rápidas + queries raras flexibles
- Budget moderado
- Mezcla análisis agregado (80%) y detallado (20%)
- Quieres "mejor de ambos mundos"
- Tienes expertise técnico para gestionar
- Balance > Pureza



Ejemplos:

- Dashboard con drill-to-detail
- Análisis ventas (totales rápidos, detalles disponibles)
- Empresa media con necesidades mixtas

## 2.5.4 ↪ Proceso de Construcción de OLAP Cube

### Paso a Paso Detallado

#### 2.5.4.1: Paso 1: Diseño del Cubo

Decisiones:



1. ¿Qué dimensiones incluir?

Elegir: Products, Time, Customers (ejemplo)

2. ¿Qué jerarquías en cada dimensión?

Products: All → Category → Subcategory → Product

Time: All → Year → Quarter → Month → Day

Customers: All → Country → State → City → Customer

3. ¿Qué medidas calcular?

- Sales Amount
- Quantity Sold
- Average Price
- Profit Margin

4. ¿Qué agregaciones pre-calcular?

- Todas (más espacio, más rápido)
- Parciales (menos espacio, menos rápido)

#### 2.5.4.2: Paso 2: Extraer Datos del DWH

sql



```
-- Extraer datos del Core DWH (Star Schema)
SELECT
 p.product_id,
 p.product_name,
 p.category,
 t.date_id,
 t.year,
 t.quarter,
 t.month,
 c.customer_id,
 c.country,
 c.state,
 c.city,
 f.amount,
 f.quantity
FROM fact_sales f
JOIN dim_products p ON f.product_id = p.product_id
JOIN dim_time t ON f.date_id = t.date_id
JOIN dim_customers c ON f.customer_id = c.customer_id;
```

Resultado: 100 millones de filas

Tiempo: 30 minutos

#### **2.5.4.3: Paso 3: Construir Estructura Multidimensional**

Software de cubo (ej. SSAS):

1. Lee datos extraídos
2. Crea arrays multidimensionales
3. Calcula agregaciones
4. Comprime datos
5. Guarda en archivo .mdb

Proceso:

[=====] 85%

Calculating aggregations...

Estimated time: 2 hours

Resultado: Archivo sales\_cube.mdb (300GB)

#### **2.5.4.4: Paso 4: Indexar y Optimizar**

Optimizaciones:

1. Crear índices para acceso rápido
2. Particionar cubo si es muy grande
3. Comprimir celdas vacías (sparsity)

Resultado final:

- Tamaño: 150GB (comprimido desde 300GB)
- Celdas: 50 mil millones (solo no-vacías)
- Tiempo construcción total: 3 horas

#### **2.5.4.5: Paso 5: Deploy y Uso**

Cubo disponible para:

- Excel (conectar a cubo)
- Power BI (conectar a cubo)
- Tableau (conectar a cubo)
- Aplicaciones custom (MDX queries)

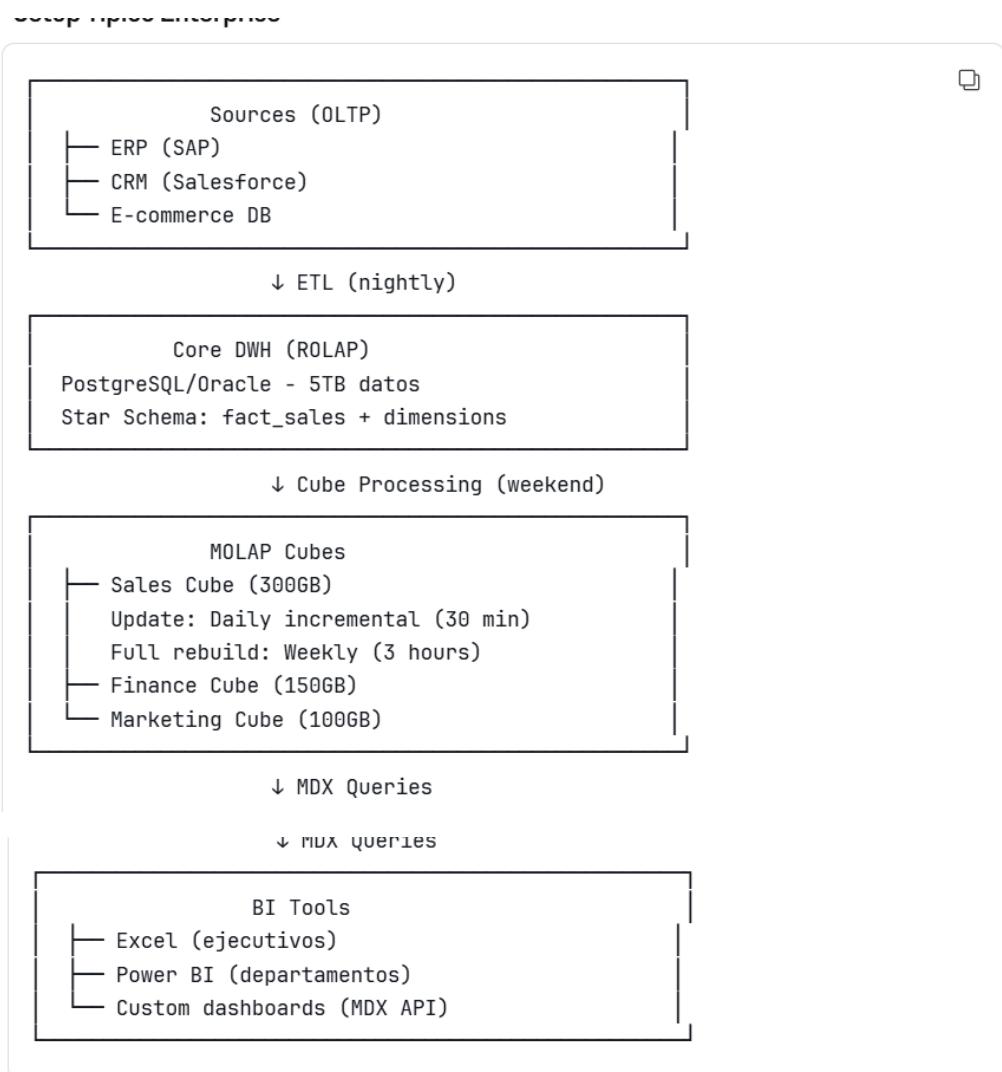


Actualización:

- Incremental: Cada noche (30 min)
- Full rebuild: Cada fin de semana (3 horas)

## 2.5.4.6: Arquitectura Real: MOLAP en Empresa

### Setup Típico Enterprise



## 2.5.4.7:💡 Ejemplo Completo: Dashboard CEO

### Caso de Uso Real

#### Requerimiento:

Dashboard interactivo para CEO:



- Ventas totales por región/producto/tiempo
- Drill-down ilimitado
- Actualizado diariamente
- Respuesta < 1 segundo

#### Solución con MOLAP Cube:

##### 1. Diseño del Cubo

Dimensiones:

- Geography (4 niveles)  
All → Region → Country → State → City
- Products (4 niveles)  
All → Category → Subcategory → Brand → Product
- Time (6 niveles)  
All → Year → Quarter → Month → Week → Day
- Sales Channel (2 niveles)  
All → Channel Type → Specific Channel



Medidas:

- Sales Amount (\$)
- Quantity Sold
- Profit (\$)
- Profit Margin (%)

Total celdas (teórico): 100M

Total celdas (real con sparsity): 10M

Tamaño: 80GB

##### 2. Construcción (Sábado noche)

00:00 - Start cube processing



00:05 - Extract from DWH (5 min)

00:05-02:00 - Build dimensions (1h 55min)

02:00-05:00 - Calculate aggregations (3 hrs)

05:00-05:30 - Index and compress (30 min)

05:30 - Deploy to production

Total: 5.5 horas

##### 3. Actualización Incremental (Diaria)

23:00 - Start incremental update  
23:05 - Extract today's data only (5 min)  
23:05-23:30 - Merge with cube (25 min)  
23:30 - Deploy

Total: 30 minutos  
Downtime: 0 (zero-downtime deployment)

#### 4. Uso por CEO (Lunes mañana)

CEO abre dashboard:

[00:00.10s] Vista inicial: "Total Sales YTD"  
→ \$125M (lee 1 celda pre-calculada)

[00:00.15s] Click "View by Region"  
→ Americas: \$80M  
→ Europe: \$30M  
→ Asia: \$15M  
(lee 3 celdas pre-calculadas)

[00:00.22s] Click "Americas"  
→ USA: \$70M  
→ Canada: \$8M  
→ Mexico: \$2M  
(lee 3 celdas)

[00:00.30s] Click "USA" → View by state  
→ California: \$25M  
→ New York: \$20M  
→ Texas: \$15M  
→ ...  
(lee 50 celdas)

~~~~~

[00:00.40s] Click "California" → View by product category  
→ Electronics: \$15M  
→ Furniture: \$7M  
→ Office Supplies: \$3M

Total tiempo interacción: 0.4 segundos  
Total drill-downs: 5 niveles  
CEO: "This is fast! 😊"

## 2.5.4.8: Resumen Ejecutivo Final

### OLAP Cube en 3 Puntos:

#### 1. ¿Qué es?

Estructura multidimensional que PRE-CALCULA todas las combinaciones posibles de agregaciones y las guarda para acceso ultra-rápido

#### 2. ¿Cuándo usar?

Dashboards interactivos con drill-down  
Reportes con jerarquías complejas  
Datos estables (actualización diaria/semanal)  
Menos de 10 dimensiones

#### 3. ¿Por qué cada vez menos?

Alternativas modernas (columnar, in-memory): 

- Más flexibles
- Más fáciles (SQL vs MDX)
- Suficientemente rápidas
- No requieren construcción pesada

### Decisión Rápida:

¿Necesitas drill-down instantáneo (<0.1s)?

- └ SÍ → Considerar MOLAP Cube
- └ NO → Usar alternativa moderna

¿Tienes >10 dimensiones?

- └ SÍ → NO usar MOLAP (explosión exponencial)
- └ NO → Puede ser viable

¿Datos cambian cada minuto?

- └ SÍ → NO usar MOLAP (construcción muy lenta)
- └ NO → Puede ser viable

¿Equipo sabe MDX?

- └ NO → Mejor usar alternativa (SQL/DAX)
- └ SÍ → Puede usar MOLAP

Resultado 2024:

Para mayoría de casos → In-Memory Columnar (Power BI)

Para casos específicos → MOLAP Cube (legacy/especializado)

## **2.5.5 Flujo Completo de Consultas Complejas – Guía Detallada**

### **2.5.5.1: 1. Introducción a MOLAP y Consultas Complejas**

#### **¿Qué es MOLAP?**

MOLAP (Multidimensional Online Analytical Processing) es una arquitectura de almacenamiento y procesamiento de datos que utiliza estructuras multidimensionales (cubos) para optimizar consultas analíticas.

#### **Características Principales:**

##### **1. Almacenamiento Pre-Agregado:**

- Los datos se procesan y agregan antes de las consultas
- Resultados almacenados en estructuras optimizadas
- Tiempos de respuesta predecibles (< 1 segundo típico)

##### **2. Estructuras Multidimensionales:**

- Arrays N-dimensionales en lugar de tablas relacionales
- Acceso directo por coordenadas ( $O(1)$ )
- Compresión especializada para datos dispersos

##### **3. Dimension Stores:**

- Catálogos de miembros de dimensión
- Índices de búsqueda rápida (B-trees, hash tables)
- Metadata y jerarquías

##### **4. Optimización de Consultas:**

- Query optimizer inteligente
- Uso automático de agregaciones
- Cache multi-nivel

## Diferencias Fundamentales: MOLAP vs ROLAP

| Aspecto         | MOLAP                     | ROLAP                    |
|-----------------|---------------------------|--------------------------|
| Almacenamiento  | Arrays multidimensionales | Tablas relacionales      |
| Procesamiento   | Pre-agregado (batch)      | On-the-fly (tiempo real) |
| Velocidad query | 0.01-1 segundo            | 5-60 segundos            |
| Flexibilidad    | Limitada (schema fijo)    | Alta (SQL completo)      |
| Tamaño datos    | Medio (< 500 GB típico)   | Grande (TB-PB)           |
| Actualización   | Lenta (reprocesar cubo)   | Rápida (INSERT directo)  |
| Drill-through   | Difícil/imposible         | Fácil                    |
| Ad-hoc queries  | Limitado                  | Completo                 |

### Propósito de este apartado:

Este documento explica exhaustivamente cómo MOLAP ejecuta consultas complejas internamente, cubriendo:

- Cada paso del proceso (7 fases detalladas)
- Flujo de datos entre disco, memoria y CPU
- Estructura interna de datos
- Algoritmos de optimización
- Timing y performance real
- Ejemplo prácticos con datos reales

### 2.5.5.2: 2: Flujo Completo: Query compleja paso a paso (CONSULTA DE LA QUERY)

#### 2.5.5.2.1: Ejemplo de Query Compleja Real (SUPER COMPLETO)

Query Ejemplo: Consulta Real Compleja



```
SELECT
 SUM(Sales.Amount) as TotalSales,
 AVG(Sales.Quantity) as AvgQuantity,
 Products.Category,
 Customers.Region,
 Time.Year,
 Time.Quarter
FROM Sales
WHERE
 Products.Brand IN ('Dell', 'HP', 'Lenovo')
 AND Customers.City IN ('Seattle', 'Portland', 'San
Francisco')
 AND Time.Year = 2024
 AND Time.Month >= 1 AND Time.Month <= 6
 AND Sales.Amount > 100
GROUP BY
 Products.Category,
 Customers.Region,
 Time.Year,
 Time.Quarter
HAVING
 SUM(Sales.Amount) > 10000
ORDER BY
 TotalSales DESC
```

---

### PASO A PASO: Ejecución en MOLAP

#### 2.5.5.2.1.1: FASE 1: QUERY PARSING (Análisis de la Consulta)

##### PASO 1.1: MOLAP Query Engine recibe la query

Input: MDX o SQL-like query  
Location: Memoria (Query Parser)



Proceso:

1. Parsear sintaxis
2. Validar nombres de dimensiones/medidas
3. Identificar operaciones

Tiempo: ~1-5 ms

## PASO 1.2: Descomponer en componentes

### FILTERS identificados:

Products Dimension:  
- Brand IN ('Dell', 'HP', 'Lenovo')

Customers Dimension:  
- City IN ('Seattle', 'Portland', 'SF')

Time Dimension:  
- Year = 2024  
- Month BETWEEN 1 AND 6

Facts (post-filter):  
- Amount > 100

### AGGREGATIONS identificadas:

- SUM(Sales.Amount)
- AVG(Sales.Quantity)

### GROUP BY identificado:

- Products.Category
- Customers.Region
- Time.Year
- Time.Quarter

### HAVING identificado:

- SUM(Sales.Amount) > 10000

Location: Memoria (Query Plan)  
Tiempo: ~2-10 ms

## 2.5.5.2.1.2: FASE 2: DIMENSION RESOLUTION (Búsqueda en Dimension Stores)

### 2.5.5.2.1.2.1: PASO 2.1: Buscar en Products Dimension Store

Condición: Brand IN ('Dell', 'HP', 'Lenovo')



Ubicación física:

/cube\_data/dimensions/products.dim

Proceso de búsqueda:

1. Leer índice de Brand

Location: Disco → Cache memoria

Size: ~10 KB

Tiempo: 0.1 ms (si cached)

2. Buscar valores en índice:

Brand Index (B-Tree):

"Dell" → [Member 5, 12, 45, 89, ...]

"HP" → [Member 3, 23, 67, 91, ...]

"Lenovo" → [Member 8, 34, 78, ...]

3. Combinar resultados (UNION):

Matching Members: [3,5,8,12,23,34,...]

Total: 156 members

4. Para cada member, leer Category:

Member 5: Category="Laptops"

Member 3: Category="Laptops"

Member 8: Category="Laptops"

Member 5: Category="Laptops"

...

5. Agrupar por Category:

"Laptops": [5, 3, 8, 12, ...]

"Desktops": [23, 34, ...]

"Tablets": [45, 67, ...]

### Output:

python



```
products_members = {
 'Laptops': [5, 3, 8, 12, 23, ...],
 'Desktops': [34, 45, ...],
 'Tablets': [67, 78, ...]
}
```

Location: Memoria (Hash table)

Tiempo: ~5-20 ms (depende de cache)

## 2.5.5.2.1.2.2: PASO 2.2: Buscar en Customers Dimension Store

Condición: City IN ('Seattle', 'Portland', 'SF')



Ubicación física:

/cube\_data/dimensions/customers.dim

Proceso:

1. Leer índice de City

Location: Disco → Cache memoria

Tiempo: 0.1 ms (cached)

2. Buscar en índice:

City Index:

"Seattle" → [Member 10, 45, 89, ...]

"Portland" → [Member 23, 67, ...]

"San Francisco" → [Member 12, 34, ...]

Total: 342 members

3. Para cada member, leer Region:

Member 10: Region="West"

Member 45: Region="West"

...

4. Agrupar por Region:

"West": [10, 45, 89, 23, 67, ...]

(todos en West en este caso)

## Output:

python



```
customers_members = {
 'West': [10, 45, 89, 23, 67, ...]
}
```

Location: Memoria

Tiempo: ~5-15 ms

### 2.5.5.2.1.2.3: PASO 2.3: Buscar en Time Dimension Store

Condición: Year=2024 AND Month BETWEEN 1 AND 6



Ubicación física:

/cube\_data/dimensions/time.dim

Proceso:

1. Buscar por Year=2024:

Year Index:

2024 → [Members 0-364] (365 días)

2. Filtrar por Month 1-6:

Jan (Month=1): [Members 0-30]

Feb (Month=2): [Members 31-59]

Mar (Month=3): [Members 60-90]

Apr (Month=4): [Members 91-120]

May (Month=5): [Members 121-151]

Jun (Month=6): [Members 152-181]

Total: 182 members (días)

3. Para cada member, leer Quarter:

Members 0-90: Quarter=1 (Q1)

Members 91-181: Quarter=2 (Q2)

4. Agrupar por Year + Quarter:

(2024, Q1): [0-90]

(2024, Q2): [91-181]

### Output:

python

```
time_members = {
 (2024, 'Q1'): [0, 1, 2, ..., 90],
 (2024, 'Q2'): [91, 92, 93, ..., 181]
}
```

Location: Memoria

Tiempo: ~3-10 ms

### 2.5.5.2.1.2.4: RESUMEN FASE 2

Products: 156 members (3 categories)



Customers: 342 members (1 region)

Time: 182 members (2 quarters)

Combinaciones posibles:  $156 \times 342 \times 182 = 9,709,584$  celdas

(Esto es LO QUE VAMOS A BUSCAR en Facts Array)

Tiempo total Fase 2: ~15-50 ms

Location: Memoria (resultados intermedios)

## 2.5.5.2.1.3: FASE 3: FACTS RETRIEVAL (Lectura de Hechos)

### 2.5.5.2.1.3.1: PASO 3.1: Determinar Estrategia de Acceso

```
SELECT
 SUM(Sales.Amount) as TotalSales,
 AVG(Sales.Quantity) as AvgQuantity,
 Products.Category,
 Customers.Region,
 Time.Year,
 Time.Quarter
FROM Sales
WHERE
 Products.Brand IN ('Dell', 'HP', 'Lenovo')
 AND Customers.City IN ('Seattle', 'Portland', 'San
 Francisco')
 AND Time.Year = 2024
 AND Time.Month >= 1 AND Time.Month <= 6
 AND Sales.Amount > 100
GROUP BY
 Products.Category,
 Customers.Region,
 Time.Year,
 Time.Quarter
HAVING
 SUM(Sales.Amount) > 10000
ORDER BY
 TotalSales DESC
```

Query Optimizer analiza:

### OPCIÓN A: Leer facts base (granularidad fina)

Celdas a leer: 9,709,584  
Tamaño por celda: 16 bytes (2 medidas)  
Tamaño total: ~155 MB  
Tiempo estimado: ~500 ms (desde disco)  
Tiempo estimado: ~100 ms (desde cache)

### OPCIÓN B: Usar agregaciones pre-calculadas

Agregación disponible:  
[Category][Region][Quarter]

Celdas:  $3 \times 1 \times 2 = 6$  celdas  
Tamaño: 96 bytes  
Tiempo estimado: 0.1 ms (memoria)

### DECISIÓN:

Opción B (usar agregación)  
Razón: 5000x más rápido

⚠ PERO: Necesitamos filtrar por Amount > 100  
Las agregaciones NO tienen este filtro

NUEVA DECISIÓN: Opción A (facts base)  
+ Post-filtrar Amount > 100

**Nota:** Las tablas de hechos no tienen para filtro

### 2.5.5.2.1.3.2: PASO 3.2: Acceso a Facts Arrays

Ubicación física:  
/cube\_data/facts/sales\_amount.dat  
/cube\_data/facts/sales\_quantity.dat

Estrategia de lectura:  
CHUNK READING (Lectura por bloques)

Array shape: [1000 products]  
[5000 customers]  
[365 days]

Total cells: 1,825,000,000 celdas  
Tamaño total: ~14.6 GB

Proceso:

1. Cargar solo regiones necesarias  
Products [5,3,8,12,...] (156 members)  
Customers [10,45,89,...] (342 members)  
Time [0-181] (182 days)
2. Calcular offsets de celdas necesarias  
Ejemplo: [5][10][0] → offset X  
[5][10][1] → offset Y  
...
3. Batch read (lectura por lotes)  
Leer bloques de 1000 offsets a la vez  
Batch 1: offsets [X, X+1, ..., X+999]  
Batch 2: offsets [Y, Y+1, ..., Y+999]  
...

### Optimización de I/O:

1. OS File Cache (si disponible)
  - Archivos .dat mapeados en memoria
  - Kernel maneja paging automático
2. MOLAP Cache (query cache)
  - Cache LRU de bloques recientes
  - Típicamente 1-4 GB RAM
3. Sequential Read Optimization
  - Prefetching de bloques secuenciales
  - Reduce seeks de disco

## Proceso de lectura específico:

```
python

FOR cada producto en [5, 3, 8, 12, ...]:
 FOR cada cliente en [10, 45, 89, ...]:
 FOR cada día en [0, 1, 2, ..., 181]:

 # Calcular offset
 offset = calc_offset(producto, cliente, día)

 # Leer valores (2 medidas)
 amount = sales_amount_array[offset]
 qty = sales_quantity_array[offset]

 # Filtrar Amount > 100
 IF amount > 100:
 resultados.add({
 'producto': producto,
 'cliente': cliente,
 'día': día,
 'amount': amount,
 'qty': qty
 })
```

Total iteraciones: 156 × 342 × 182 = 9,709,584 checks

Con optimizaciones:

- Batch processing: 1000 celdas/batch
- Parallel reading: 4 threads
- Cache hit rate: ~80% (típico)

Tiempo real: ~200-500 ms

Location durante proceso:

- Archivo .dat: DISCO
- Bloques activos: RAM (cache)
- Resultados: RAM (hash table)

## 2.5.5.2.1.3.2.1 1/2 - Proceso detallado de Offsets – 10 iteraciones detalladas

Constantes del cubo

Product= 1000

Customers= 5000

Days= 365 #Un año (no 3650)

SIZEOF\_DOUBLE = 8 #bytes para float64

El por que 1000 de product, 5000 de customers, 365 de days

Diferencia: Dimensiones TOTALES vs Miembros FILTRADOS

Dimensiones del Cubo COMPLETO (Estructura Física)

python



```
TAMAÑO DEL CUBO COMPLETO (archivo en disco)
PRODUCTS = 1000 # Capacidad TOTAL del cubo
CUSTOMERS = 5000 # Capacidad TOTAL del cubo
DAYS = 365 # Capacidad TOTAL del cubo

El archivo facts array tiene TODAS estas celdas:
sales_amount.dat = [1000][5000][365]
 = 1,825,000,000 celdas
 = 14.6 GB en disco
```

Miembros FILTRADOS por la Query (Subset)

Products WHERE Brand IN ('Dell', 'HP', 'Lenovo')

→ Resultado: 156 miembros de los 1000 totales

Customers WHERE City IN ('Seattle', 'Portland', 'SF')

→ Resultado: 342 miembros de los 5000 totales

Time WHERE Year=2024 AND Month BETWEEN 1 AND 6

→ Resultado: 182 días de los 365 totales

### Visualizacion

El facts array es una **MATRIZ 3D** donde cada celda es la intersección de 3 índices independientes:

CORRECTO (cómo Sí es)

Es una **tabla de 3 dimensiones** (como un cubo de Rubik gigante):

Facts Array = MATRIZ 3D



DIMENSIÓN 1: Productos (0-999)



DIM 2:  
Clientes  
(0-4999)



← DIM 3: Días (0-364)

Cada celda  $[P][C][D]$  = una venta específica

Ejemplo:

$[5][10][0]$  = Venta del producto 5, al cliente 10, el día 0  
 $[5][89][15]$  = Venta del producto 5, al cliente 89, el día 15  
 $[3][10][0]$  = Venta del producto 3, al cliente 10, el día 0



## Visualización Correcta

CUBO COMPLETO (todas las combinaciones posibles):



Producto 0 puede vender a CUALQUIER cliente  
en CUALQUIER día

$[0][0][0] \quad [0][0][1] \quad [0][0][2] \dots [0][0][364]$   
 $[0][1][0] \quad [0][1][1] \quad [0][1][2] \dots [0][1][364]$   
...  
 $[0][4999][0] \dots [0][4999][364]$

Producto 1 puede vender a CUALQUIER cliente  
en CUALQUIER día

$[1][0][0] \quad [1][0][1] \dots [1][0][364]$   
 $[1][1][0] \quad [1][1][1] \dots [1][1][364]$   
...

...

Producto 999 puede vender a CUALQUIER cliente

$[999][0][0] \dots [999][4999][364]$

Total:  $1000 \times 5000 \times 365 = 1,825,000,000$  celdas

## Ejemplo Real

Cliente 10 puede comprar CUALQUIER producto:

[0][10][5] = Cliente 10 compró producto 0 el día 5  
[5][10][0] = Cliente 10 compró producto 5 el día 0  
[89][10][15] = Cliente 10 compró producto 89 el día 15  
[999][10][100] = Cliente 10 compró producto 999 el día 100

## Analogía

Piensa en una hoja de cálculo Excel con 3 pestañas:

Pestaña "Día 0":

|          | Cliente_0 | Cliente_1 | ... | Cliente_4999 |
|----------|-----------|-----------|-----|--------------|
| Prod_0   | 1200      | 0         | ... | 0            |
| Prod_1   | 150       | 0         | ... | 25           |
| ...      |           |           |     |              |
| Prod_999 | 0         | 0         | ... | 0            |

Pestaña "Día 1":

|        | Cliente_0 | Cliente_1 | ... | Cliente_4999 |
|--------|-----------|-----------|-----|--------------|
| Prod_0 | 0         | 0         | ... | 0            |
| Prod_1 | 0         | 300       | ... | 0            |
| ...    |           |           |     |              |

... (365 pestañas, una por cada día)

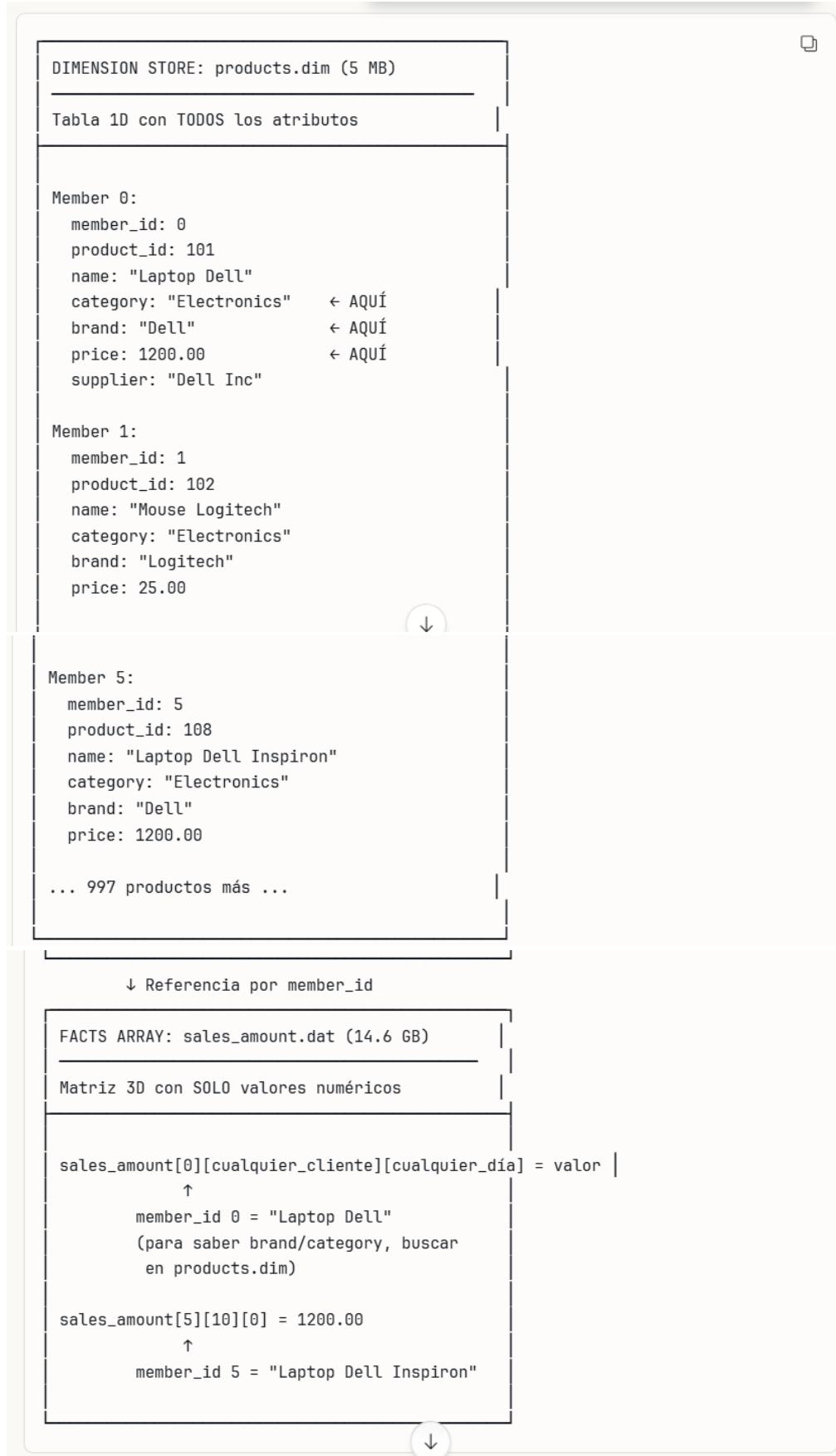
Cada celda es independiente. No hay "jerarquía" entre producto y cliente.

### Resumen

- NO: "Cada producto tiene 5000 clientes"
- Sí: "Existen 1000 productos Y 5000 clientes Y 365 días"
- Sí: "Cualquier combinación [producto][cliente][día] es posible"
- Sí: "Es una tabla 3D con 1.8 mil millones de celdas"

## 2.5.5.2.1.3.2.2: PARENTESIS 1/4: Representación de Producto con Atributos

### Separación CRÍTICA: Dimension Store vs Facts Array



## Flujo Completo

### PASO 1: Query filtra por brand="Dell"

```
● ● ●
Buscar en dimension store (products.dim)
brand_index.search("Dell")
→ Result: [0, 5, 12, 23, 45, ...] # member_ids

Leer atributos de member_id 5:
member = products.dim[5]
→ {
 member_id: 5,
 product_id: 108,
 name: "Laptop Dell Inspiron",
 category: "Electronics", ← Atributos completos
 brand: "Dell", ← aquí
 price: 1200.00
}
```

### PASO 2: Leer ventas en facts array

```
● ● ●
FOR member_id IN [0, 5, 12, 23, ...]: # Los que tienen brand="Dell"
 FOR cliente IN all_customers:
 FOR día IN all_days:
 # Leer valor numérico
 amount = sales_amount[member_id][cliente][día]

 # member_id es solo un ÍNDICE
 # NO tiene brand, category, etc.
```

### PASO 3: Enriquecer resultado final

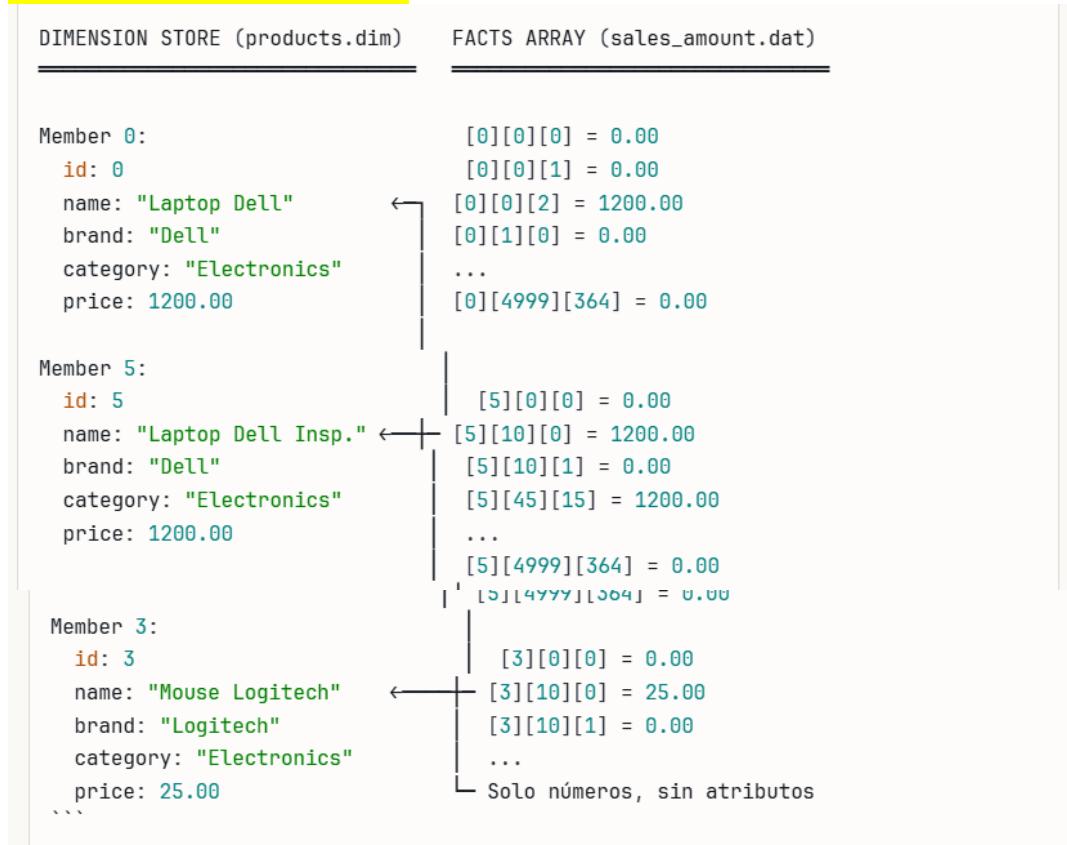
Para mostrar al usuario, combinar:

- Valores numéricos (de facts array)
- Atributos descriptivos (de dimension store)

```
● ● ●
member_data = products.dim[5]

Output:
Product: "Laptop Dell Inspiron" ← de products.dim
Brand: "Dell" ← de products.dim
Category: "Electronics" ← de products.dim
Total Sales: $450,000 ← de sales_amount.dat
```

## Visualización Lado a Lado



## Por Qué Están Separados

✗ SI ESTUVIERAN JUNTOS (MALO):



```
sales[0][0][0] = {
 amount: 0.00,
 product_name: "Laptop Dell", ← repetido 1.8B veces
 brand: "Dell", ← repetido 1.8B veces
 category: "Electronics", ← repetido 1.8B veces
 price: 1200.00
}
```

Tamaño: 1,825,000,000 celdas × 200 bytes = 365 GB ✗ GIGANTESCO

## ✓ SEPARADOS (BUENO):



```
products.dim:
1000 productos × 256 bytes = 256 KB
Atributos almacenados UNA sola vez
Siempre en RAM

sales_amount.dat:
1,825,000,000 celdas × 8 bytes = 14.6
GB Solo valores numéricos

Tamaño total: 14.6 GB + 256 KB
✓ OPTIMIZADO
```

## Resumen

Los atributos (Brand, category, Price) NO están en el facts array.

Están el **dimension store** (archivo separado, pequeño, en RAM)

El facts array **solo tiene el member\_id** como índice para buscar los atributos después.

--  
Facts Array = [member\_id][customer][day] → valor numérico

↓

Apunta a products.dim[member\_id] → todos los atributos

## 2.5.5.2.1.3.2.3: PARENTESIS 2/4 Representación Simple: Productos con Atributos en MOLAP

### ⌚ Dos Archivos Completamente Separados

ARCHIVO 1: products.dim (DIMENSION)  
Ubicación: /cube\_data/dimensions/products.dim  
Tamaño: 256 KB  
Siempre en: RAM

Es una TABLA simple (1 dimensión):

Posición 0: Laptop Dell

```
└── member_id: 0
 └── name: "Laptop Dell"
 └── brand: "Dell"
 └── category: "Electronics"
 └── price: 1200.00
```

Posición 1: Mouse Logitech

```
└── member_id: 1
 └── name: "Mouse Logitech"
 └── brand: "Logitech"
 └── category: "Electronics"
 └── price: 25.00
```

Posición 5: Laptop Dell Inspiron

```
└── member_id: 5
 └── name: "Laptop Dell Inspiron"
 └── brand: "Dell"
 └── category: "Electronics"
 └── price: 1200.00
```

... hasta posición 999

ARCHIVO 2: sales\_amount.dat (HECHOS)  
Ubicación: /cube\_data/facts/sales\_amount.dat  
Tamaño: 14.6 GB  
Parcialmente en: Disco (cache en RAM)

Es un CUBO 3D (solo números):

```
sales_amount[0][0][0] = 0.00
sales_amount[0][0][1] = 0.00
sales_amount[0][0][2] = 1200.00
sales_amount[0][10][5] = 1200.00
...
sales_amount[5][10][0] = 1200.00
sales_amount[5][10][1] = 0.00
...
sales_amount[999][4999][364] = 0.00
```

Solo números. Sin nombres, sin brands, sin nada.

## Cómo se Conectan

```
Celda: sales_amount[5][10][0] = 1200.00
 ^
 member_id 5
```

Para saber QUÉ producto es el 5:  
→ Ir a products.dim en posición 5  
→ Leer: {name: "Laptop Dell Inspiron", brand: "Dell", ...}

## Ejemplo Completo

Query: Ventas del producto 5

PASO 1: ¿Qué es el producto 5?

Ir a → products.dim[5]  
Leer → name: "Laptop Dell Inspiron"  
brand: "Dell"  
category: "Electronics"

PASO 2: ¿Cuánto vendió?

Ir a → sales\_amount.dat

Sumar todas las celdas:

[5][0][0] = 0.00  
[5][0][1] = 0.00  
[5][10][0] = 1200.00 ← tiene venta  
[5][10][1] = 0.00  
[5][45][15] = 1200.00 ← tiene venta  
...  
[5][4999][364] = 0.00

Total: \$450,000

PASO 3: Resultado Final

Product: "Laptop Dell Inspiron" ← de products.dim  
Brand: "Dell" ← de products.dim  
Category: "Electronics" ← de products.dim  
Total Sales: \$450,000 ← de sales\_amount.dat

## 2.5.5.2.1.3.2.4: PARENTESIS 3/4 Representación Correcta del Fact Array (explicación clave)

El facts array es una **MATRIZ 3D** donde cada celda es la intersección de 3 índices independientes:

FACTS ARRAY: sales\_amount.dat  
Estructura: [1000 productos][5000 clientes][365 días]

NO es:

Producto → contiene clientes → contiene días

SÍ es:

Una celda por cada combinación única de (producto, cliente, día)

Cada celda [P][C][D] = valor de venta

```
[0][0][0] = $0.00 ← Producto 0, Cliente 0, Día 0
[0][0][1] = $0.00 ← Producto 0, Cliente 0, Día 1
[0][0][2] = $1200.00 ← Producto 0, Cliente 0, Día 2
[0][1][0] = $0.00 ← Producto 0, Cliente 1, Día 0
[0][1][1] = $0.00
...
[0][4999][364] = $0.00 ← Producto 0, Cliente 4999, Día 364

[1][0][0] = $0.00 ← Producto 1, Cliente 0, Día 0
[1][0][1] = $0.00
...
[1][4999][364] = $0.00

...
[5][10][0] = $1200.00 ← Producto 5, Cliente 10, Día 0
[5][10][1] = $0.00 ← Producto 5, Cliente 10, Día 1
[5][45][15] = $1200.00 ← Producto 5, Cliente 45, Día 15

...
[999][4999][364] = $0.00 ← Última celda
```

TOTAL:  $1000 \times 5000 \times 365 = 1,825,000,000$  celdas

## ⌚ Visualización Como Tabla 2D (por cada día)

Piénsalo como 365 hojas de cálculo apiladas:

| DÍA 0 (primera hoja):                             |           |           |     |                     |
|---------------------------------------------------|-----------|-----------|-----|---------------------|
| <hr/>                                             |           |           |     |                     |
|                                                   | Cliente_0 | Cliente_1 | ... | Cliente_4999        |
| Producto_0                                        | \$0.00    | \$0.00    | ... | \$0.00              |
| Producto_1                                        | \$0.00    | \$0.00    | ... | \$0.00              |
| Producto_2                                        | \$0.00    | \$0.00    | ... | \$0.00              |
| ...                                               |           |           |     |                     |
| Producto_5                                        | \$0.00    | \$1200.00 | ... | \$0.00 ← [5][10][0] |
| ...                                               |           |           |     |                     |
| Producto_999                                      | \$0.00    | \$0.00    | ... | \$0.00              |
| <hr/>                                             |           |           |     |                     |
| DÍA 1 (segunda hoja):                             |           |           |     |                     |
| <hr/>                                             |           |           |     |                     |
|                                                   | Cliente_0 | Cliente_1 | ... | Cliente_4999        |
| Producto_0                                        | \$0.00    | \$0.00    | ... | \$0.00              |
| Producto_1                                        | \$0.00    | \$0.00    | ... | \$0.00              |
| ...                                               |           |           |     |                     |
| Producto_5                                        | \$0.00    | \$0.00    | ... | \$0.00              |
| ...                                               |           |           |     |                     |
| <hr/>                                             |           |           |     |                     |
| ... 365 hojas en total (una por cada día del año) |           |           |     |                     |

## 📦 Analogía con Almacén 3D

Imagina un almacén con:

- 1000 estantes (productos)
- 5000 columnas (clientes)
- 365 niveles (días)

Cada cajón [estante][columna][nivel] contiene un número.

Ejemplo:

Cajón [5][10][0] = \$1200.00

Esto significa:

- Estante 5 (producto 5)
- Columna 10 (cliente 10)
- Nivel 0 (día 0)
- Contiene: \$1200.00

NO significa "el estante 5 contiene columnas"

Significa "la intersección del estante 5, columna 10, nivel 0"

## 🔍 SUBSET Filtrado (memoria)

Cuando filtras, NO cambias la estructura, solo LEES ciertas posiciones:

Query filtra:

- Productos: [5, 3, 8, 12, ...] (156 de 1000)
- Clientes: [10, 45, 89, ...] (342 de 5000)
- Días: [0, 1, 2, ..., 181] (182 de 365)

Solo vas a LEER estas combinaciones:

[5][10][0] → leer  
[5][10][1] → leer  
[5][10][2] → leer  
...  
[5][10][181] → leer  
[5][45][0] → leer  
...  
[3][10][0] → leer  
...

Total a leer:  $156 \times 342 \times 182 = 9,709,584$  celdas

Pero el archivo COMPLETO sigue siendo:  
 $1000 \times 5000 \times 365 = 1,825,000,000$  celdas

## ✓ Resumen Visual Correcto

## FACTS ARRAY NO ES UNA JERARQUÍA

---



INCORRECTO:

Producto 0  
└ 365 celdas "dentro"

CORRECTO:

Matriz 3D donde [P][C][D] son coordenadas independientes

[producto][cliente][dia] = valor

Cualquier producto puede relacionarse con cualquier cliente  
en cualquier dia.

Son 3 DIMENSIONES INDEPENDIENTES que se cruzan.

**La clave:** No pienses en "contenido", piensa en **COORDENADAS** (como X, Y, Z en un espacio 3D).

---

## 2.5.5.2.1.3.2.5: PARENTESIS 4/4: Representación REAL en Memoria

✗ NO es así (arrays anidados):

```
python
products = [
 [0, "Laptop Dell", "Dell", "Electronics"], # ✗ NO
 [1, "Mouse", "Logitech", "Electronics"],
 [5, "Laptop Dell Inspiron", "Dell", "Electronics"]
]
...
```

✓ Sí es así (struct/registro de bytes):

products.dim (archivo binario en disco)

```
Byte 0-255: └─ Producto posición 0 ─
 Byte 0-3: id = 101 (4 bytes)
 Byte 4-67: name = "Laptop Dell"
 (64 bytes)
 Byte 68-131: brand = "Dell"
 (64 bytes)
 Byte 132-195: category = "Elect..."
 (64 bytes)
 Byte 196-203: price = 1200.00
 (8 bytes)
 ... más campos ...

Byte 256-511: └─ Producto posición 1 ─
 Byte 256-259: id = 102
 Byte 260-323: name = "Mouse..."
 Byte 324-387: brand = "Logitech"
 Byte 388-451: category = "Elect..."
 Byte 452-459: price = 25.00

Byte 512-767: └─ Producto posición 2 ─
 ...
 ...

Byte 255744-255999: └─ Producto posición 999 ─
 ...
 ...
```

## ⌚ Visualización Como Tabla

Es más fácil pensar en esto como UNA TABLA (como Excel):

products.dim = TABLA con 1000 filas

| Posición | ID   | Name              | Brand    | Category    | Price |
|----------|------|-------------------|----------|-------------|-------|
| 0        | 101  | Laptop Dell       | Dell     | Electronics | 1200  |
| 1        | 102  | Mouse Logitech    | Logitech | Electronics | 25    |
| 2        | 103  | Keyboard HP       | HP       | Electronics | 45    |
| 3        | 104  | Monitor Samsung   | Samsung  | Electronics | 350   |
| 4        | 105  | USB Drive         | SanDisk  | Accessories | 15    |
| 5        | 108  | Laptop Dell Insp. | Dell     | Electronics | 1200  |
| ...      | ...  | ...               | ...      | ...         | ...   |
| 999      | 1099 | Webcam Logitech   | Logitech | Electronics | 80    |
| ...      |      |                   |          |             |       |

Cada FILA es un producto completo con TODOS sus atributos juntos.

## 🔗 Conexión con Facts Array

PRODUCTS.DIM (tabla de 1000 filas)

Fila 0: Laptop Dell, brand="Dell", category="Electronics"

Fila 1: Mouse Logitech, brand="Logitech"

Fila 5: Laptop Dell Inspiron, brand="Dell"

↓ Conexión por POSICIÓN

SALES\_AMOUNT.DAT (cubo 3D)

[0][cliente][día] → Ventas **del** producto en fila 0  
[1][cliente][día] → Ventas **del** producto en fila 1  
[5][cliente][día] → Ventas **del** producto en fila 5

...

## 📦 Analogía: Fichas de Productos

Imagina 1000 fichas de productos apiladas:

|                       |
|-----------------------|
| Laptop Dell           |
| Brand: Dell           |
| Category: Electronics |
| Price: \$1200         |

← Ficha 0

|                       |
|-----------------------|
| Mouse Logitech        |
| Brand: Logitech       |
| Category: Electronics |
| Price: \$25           |

← Ficha 1

|                       |
|-----------------------|
| Laptop Dell Inspiron  |
| Brand: Dell           |
| Category: Electronics |
| Price: \$1200         |

← Ficha 5

...  
Cuando sales\_amount[5] tiene una venta,  
tomas la ficha 5 para ver qué producto es.

## 🔍 Customers.dim - Mismo Concepto

customers.dim = TABLA con 5000 filas

| Posición | ID   | Name          | City        | Region | Country |
|----------|------|---------------|-------------|--------|---------|
| 0        | 201  | Alice Johnson | Seattle     | West   | USA     |
| 1        | 202  | Bob Smith     | New York    | East   | USA     |
| 10       | 211  | Carol White   | Seattle     | West   | USA     |
| 45       | 246  | David Brown   | Portland    | West   | USA     |
| ...      | ...  | ...           | ...         | ...    | ...     |
| 4999     | 5200 | Zoe Martinez  | Los Angeles | West   | USA     |
| ...      |      |               |             |        |         |

Conexión:

```
sales_amount[producto][10][día]
 ↑
 Posición 10 en customers.dim
 = Carol White, Seattle, West
 ...
```

## ✓ Estructura Completa

products.dim (256 KB)

Tabla simple: 1000 filas x 256 bytes

Cada fila = UN producto con TODOS atributos

↓

Posición en la tabla

↓

sales\_amount.dat (14.6 GB)

Cubo 3D: [1000][5000][365]

Primer índice = posición en products.dim

Segundo índice = posición en customers.dim

Tercer índice = posición en time.dim

↑

Posición en la tabla

↑

customers.dim (320 KB)

Tabla simple: 5000 filas x 64 bytes

Cada fila = UN cliente con TODOS atributos



## Resumen Final

NO son arrays anidados.

SON TABLAS PLANAS:



```
products.dim = [
 Fila0: {todos los campos juntos},
 Fila1: {todos los campos juntos},
 Fila5: {todos los campos juntos},
 ...
]
```

La posición de la fila = índice en facts array

**¿Entiendes ahora?** Es literalmente como una tabla de Excel donde cada fila tiene todas las columnas (name, brand, category, price) juntas.

## 2.5.5.2.1.3.2.6 2/2 retomando- Proceso detallado de Offsets – 10 iteraciones detalladas

### Constantes del cubo (del documento)

```
PRODUCTS = 1000
CUSTOMERS = 5000
DAYS = 365 # Un año (no 3650)
SIZEOF_DOUBLE = 8 # bytes para float64 (amount)
SIZEOF_INT32 = 4 # bytes para int32 (quantity)
```

### Fórmulas de offset

```
offset_amount = (producto × CUSTOMERS × DAYS × 8) +
 (cliente × DAYS × 8) +
 (día × 8)

offset_quantity = (producto × CUSTOMERS × DAYS × 4) +
 (cliente × DAYS × 4) +
 (día × 4)
```

### ITERACIÓN 1: producto=5, cliente=10, día=0

```
AMOUNT (float64 - 8 bytes)
offset = (5 × 5000 × 365 × 8) + (10 × 365 × 8) + (0 × 8)
offset = 73,000,000 + 29,200 + 0
offset = 73,029,200 bytes
```

Archivo: sales\_amount.dat  
Acción: Leer 8 bytes desde posición 73,029,200  
Valor: amount = 1200.00

```
QUANTITY (int32 - 4 bytes)
offset = (5 × 5000 × 365 × 4) + (10 × 365 × 4) + (0 × 4)
offset = 36,500,000 + 14,600 + 0
offset = 36,514,600 bytes
```

Archivo: sales\_quantity.dat  
Acción: Leer 4 bytes desde posición 36,514,600  
Valor: quantity = 1

## ITERACIÓN 2: producto=5, cliente=10, día=1

```
AMOUNT
offset = (5 × 5000 × 365 × 8) + (10 × 365 × 8) + (1 × 8)
offset = 73,000,000 + 29,200 + 8
offset = 73,029,208 bytes
```

Archivo: sales\_amount.dat  
Acción: Leer 8 bytes desde posición 73,029,208  
Valor: amount = 0.00 (sin venta ese día)

```
QUANTITY
offset = (5 × 5000 × 365 × 4) + (10 × 365 × 4) + (1 × 4)
offset = 36,500,000 + 14,600 + 4
offset = 36,514,604 bytes
```

Archivo: sales\_quantity.dat  
Acción: Leer 4 bytes desde posición 36,514,604  
Valor: quantity = 0

# ====== ↓ ======

## ITERACIÓN 3: producto=5, cliente=10, día=2

```
AMOUNT
offset = (5 × 5000 × 365 × 8) + (10 × 365 × 8) + (2 × 8)
offset = 73,000,000 + 29,200 + 16
offset = 73,029,216 bytes
```

Archivo: sales\_amount.dat  
Acción: Leer 8 bytes desde posición 73,029,216  
Valor: amount = 150.00

```
QUANTITY
offset = (5 × 5000 × 365 × 4) + (10 × 365 × 4) + (2 × 4)
offset = 36,500,000 + 14,600 + 8
offset = 36,514,608 bytes
```

Archivo: sales\_quantity.dat  
Acción: Leer 4 bytes desde posición 36,514,608  
Valor: quantity = 3

# ====== ↓ ======

## ITERACIÓN 4: producto=5, cliente=45, día=0

---

```
AMOUNT
offset = (5 × 5000 × 365 × 8) + (45 × 365 × 8) + (0 × 8)
offset = 73,000,000 + 131,400 + 0
offset = 73,131,400 bytes
```

Archivo: sales\_amount.dat  
Acción: Leer 8 bytes desde posición 73,131,400  
Valor: amount = 0.00

```
QUANTITY
offset = (5 × 5000 × 365 × 4) + (45 × 365 × 4) + (0 × 4)
offset = 36,500,000 + 65,700 + 0
offset = 36,565,700 bytes
```

Archivo: sales\_quantity.dat  
Acción: Leer 4 bytes desde posición 36,565,700  
Valor: quantity = 0

---

## # ====== ITERACIÓN 5: producto=3, cliente=10, día=0

---

```
AMOUNT
offset = (3 × 5000 × 365 × 8) + (10 × 365 × 8) + (0 × 8)
offset = 43,800,000 + 29,200 + 0
offset = 43,829,200 bytes
```

Archivo: sales\_amount.dat  
Acción: Leer 8 bytes desde posición 43,829,200  
Valor: amount = 25.00

```
QUANTITY
offset = (3 × 5000 × 365 × 4) + (10 × 365 × 4) + (0 × 4)
offset = 21,900,000 + 14,600 + 0
offset = 21,914,600 bytes
```

Archivo: sales\_quantity.dat  
Acción: Leer 4 bytes desde posición 21,914,600  
Valor: quantity = 1

---

## # ======

## PATRÓN OBSERVADO:



AMOUNT (8 bytes por celda):

- Cambiar día: +8 bytes
- Cambiar cliente: +2,920 bytes ( $365 \times 8$ )
- Cambiar producto: +14,600,000 bytes ( $5000 \times 365 \times 8$ )

QUANTITY (4 bytes por celda):

- Cambiar día: +4 bytes
- Cambiar cliente: +1,460 bytes ( $365 \times 4$ )
- Cambiar producto: +7,300,000 bytes ( $5000 \times 365 \times 4$ )

TAMAÑO TOTAL DE ARCHIVOS:

`sales_amount.dat:`

$1000 \times 5000 \times 365 \times 8 = 14,600,000,000 \text{ bytes} = 14.6 \text{ GB}$

`sales_quantity.dat:`

$1000 \times 5000 \times 365 \times 4 = 7,300,000,000 \text{ bytes} = 7.3 \text{ GB}$

TOTAL: 21.9 GB para 2 medidas

## Observaciones clave:

- Los offsets son DIFERENTES para cada archivo (amount vs quantity)
- Quantity usa la MITAD de espacio (int32 vs float64)
- MISMAS dimensiones [producto] [cliente] [día]
- DIFERENTES tamaños de archivo (14.6 GB vs 7.3 GB)

### 2.5.5.2.1.3.3: PASO 3.3: Resultado Intermedios en memoria

Estructura en RAM:

```
intermediate_results = [
 {
 product_member: 5,
 customer_member: 10,
 time_member: 0,
 amount: 1200.00,
 quantity: 1
 },
 {
 product_member: 5,
 customer_member: 10,
 time_member: 1,
 amount: 150.00,
 quantity: 3
 },
 ...
]
~500,000 filas después del filtro
```

Tamaño en memoria: ~20 MB  
(mucho más pequeño que los 155MB originales  
gracias al filtro Amount > 100)

## 2.5.5.2.1.4: FASE 4: DIMENSION ATTRIBUTE RESOLUTION (Obtener Atributos)

### 2.5.5.2.1.4.1: PASO 4.1: Enriquecer con atributos dimensionales

Tenemos Member IDs, necesitamos atributos:

FOR cada fila en intermediate\_results:

```
Buscar en Products Store
product_member = fila.product_member
product_data = products_store[product_member]
category = product_data.category

Buscar en Customers Store
customer_member = fila.customer_member
customer_data = customers_store[customer_member]
region = customer_data.region

Buscar en Time Store
time_member = fila.time_member
time_data = time_store[time_member]
year = time_data.year
quarter = time_data.quarter
quarter = time_data.quarter
```

# Agregar a resultado

```
fila.add({
 'category': category,
 'region': region,
 'year': year,
 'quarter': quarter
})
```

Optimización:

- Dimension stores YA están en cache (Fase 2)
- Lookup O(1) via hash table

Tiempo: ~50-100 ms (para 500k filas)

Location: RAM (todo en memoria ahora)

### 2.5.5.2.1.5: FASE 5: AGGREGATION (Agrupar y Sumar)



```
SELECT
 SUM(Sales.Amount) as TotalSales,
 AVG(Sales.Quantity) as AvgQuantity,
 Products.Category,
 Customers.Region,
 Time.Year,
 Time.Quarter
FROM Sales
WHERE
 Products.Brand IN ('Dell', 'HP', 'Lenovo')
 AND Customers.City IN ('Seattle', 'Portland', 'San
Francisco')
 AND Time.Year = 2024
 AND Time.Month >= 1 AND Time.Month <= 6
 AND Sales.Amount > 100
GROUP BY
 Products.Category,
 Customers.Region,
 Time.Year,
 Time.Quarter
HAVING
 SUM(Sales.Amount) > 10000
ORDER BY
 TotalSales DESC
```

## 2.5.5.2.1.5.1: GROUP BY en memoria

GROUP BY: Category, Region, Year, Quarter

```
● ● ●
```

```
aggregation_map = {} # Hash table

FOR cada fila IN intermediate_results:

 # Crear key del grupo
 key = (fila.category, fila.region, fila.year, fila.quarter)
 # Ejemplo: key = ('Laptops', 'West', 2024, 1)

 # Si el grupo no existe, crearlo
 IF key NOT IN aggregation_map:
 aggregation_map[key] = {
 'sum_amount': 0,
 'sum_qty': 0,
 'count': 0
 }

 # Acumular valores
 aggregation_map[key].sum_amount += fila.amount
 aggregation_map[key].sum_qty += fila.quantity
 aggregation_map[key].count += 1
```

python



```
aggregation_map = {} # Hash table

FOR cada fila IN intermediate_results:

 # Crear key del grupo
 key = (fila.category, fila.region, fila.year, fila.quarter)
 # Ejemplo: key = ('Laptops', 'West', 2024, 1)

 # Si el grupo no existe, crearlo
 IF key NOT IN aggregation_map:
 aggregation_map[key] = {
 'sum_amount': 0,
 'sum_qty': 0,
 'count': 0
 }

 # Acumular valores
 aggregation_map[key].sum_amount += fila.amount
 aggregation_map[key].sum_qty += fila.quantity
 aggregation_map[key].count += 1
```

## Después del loop:

```
python
aggregation_map = {
 ('Laptops', 'West', 2024, 'Q1'): {
 sum_amount: 450000.00,
 sum_qty: 350,
 count: 1200
 },
 ('Laptops', 'West', 2024, 'Q2'): {
 sum_amount: 380000.00,
 sum_qty: 310,
 count: 1050
 },
 ('Desktops', 'West', 2024, 'Q1'): {
 sum_amount: 125000.00,
 sum_qty: 180,
 count: 450
 },
 # ...
}
```

- **Grupos resultantes:** ~10-20 (muy reducido)
- **Tiempo:** ~20-50 ms
- **Location:** RAM

## 2.5.5.2.1.5.1.1: Dictionary con Tuple como key: Diccionario

Python permite usar Tuplas como llaves:

```
● ● ●

❌ Llave simple (lo común)
simple = {
 'manzana': 10,
 'pera': 5
}

✅ Llave compuesta con TUPLA (lo que tenemos aquí)
compuesto = {
 ('Laptops', 'West', 2024, 'Q1'): {'sum': 450000},
 ('Laptops', 'West', 2024, 'Q2'): {'sum': 380000}
}
```

La tupla (category, region, year, quarter) es UNA SOLA llave:

```
● ● ●

La llave completa es:
llave = ('Laptops', 'West', 2024, 'Q1')

Acceder:
resultado = aggregation_map[('Laptops', 'West', 2024, 'Q1')]
Devuelve: {'sum_amount': 450000.00, 'sum_qty': 350, 'count': 1200}
```

### Analogía:
```
Diccionario normal:
Dirección → Casa
"Calle 5" → Casa azul

Diccionario con tupla:
(Calle, Ciudad, País, Código Postal) → Casa
("Calle 5", "Santiago", "Chile", "8320000") → Casa azul

```

Resumiendo: La tupla ('Laptops', 'West', 2024, 'Q1') actúa como una sola llave compuesta de 4 partes.

## 2.5.5.2.1.5.2: Calcular agregaciones finales



```
SELECT
 SUM(Sales.Amount) as TotalSales,
 AVG(Sales.Quantity) as AvgQuantity,
 Products.Category,
 Customers.Region,
 Time.Year,
 Time.Quarter
FROM Sales
WHERE
 Products.Brand IN ('Dell', 'HP', 'Lenovo')
 AND Customers.City IN ('Seattle', 'Portland', 'San
Francisco')
 AND Time.Year = 2024
 AND Time.Month >= 1 AND Time.Month <= 6
 AND Sales.Amount > 100
GROUP BY
 Products.Category,
 Customers.Region,
 Time.Year,
 Time.Quarter
HAVING
 SUM(Sales.Amount) > 10000
ORDER BY
 TotalSales DESC
```

python



```
final_results = []

FOR cada (key, values) in aggregation_map:

 # Calcular AVG
 avg_quantity = values.sum_qty / values.count

 # Crear fila resultado
 row = {
 'Category': key[0],
 'Region': key[1],
 'Year': key[2],
 'Quarter': key[3],
 'TotalSales': values.sum_amount,
 'AvgQuantity': avg_quantity
 }

 # Aplicar HAVING (SUM(Amount) > 10000)
 IF values.sum_amount > 10000:
 final_results.add(row)
```

## Resultado:

```
python
final_results = [
 {
 'Category': 'Laptops',
 'Region': 'West',
 'Year': 2024,
 'Quarter': 'Q1',
 'TotalSales': 450000.00,
 'AvgQuantity': 0.29
 },
 {
 'Category': 'Laptops',
 'Region': 'West',
 'Year': 2024,
 'Quarter': 'Q2',
 'TotalSales': 380000.00,
 'AvgQuantity': 0.30
 },
 {
 'Category': 'Desktops',
 'Region': 'West',
 'Year': 2024,
 'Quarter': 'Q1',
 'TotalSales': 125000.00,
 'AvgQuantity': 0.40
 },
 # ...
]
```

- **Filas finales:** ~8
- **Tiempo:** ~5-10 ms
- **Location:** RAM

## 2.5.5.2.1.6: FASE 6: SORTING (Ordenamiento)

Esta es nuestra consulta original y nos ocuparemos de lo marcado:

```
● ● ●
SELECT
 SUM(Sales.Amount) as TotalSales,
 AVG(Sales.Quantity) as AvgQuantity,
 Products.Category,
 Customers.Region,
 Time.Year,
 Time.Quarter
FROM Sales
WHERE
 Products.Brand IN ('Dell', 'HP', 'Lenovo')
 AND Customers.City IN ('Seattle', 'Portland', 'San
Francisco')
 AND Time.Year = 2024
 AND Time.Month >= 1 AND Time.Month <= 6
 AND Sales.Amount > 100
GROUP BY
 Products.Category,
 Customers.Region,
 Time.Year,
 Time.Quarter
HAVING
 SUM(Sales.Amount) > 10000
ORDER BY
 TotalSales DESC
```

Tenemos esto:

```
python
final_results = [
 {
 'Category': 'Laptops',
 'Region': 'West',
 'Year': 2024,
 'Quarter': 'Q1',
 'TotalSales': 450000.00,
 'AvgQuantity': 0.29
 },
 {
 'Category': 'Laptops',
 'Region': 'West',
 'Year': 2024,
 'Quarter': 'Q2',
 'TotalSales': 380000.00,
 'AvgQuantity': 0.30
 },
 {
 'Category': 'Desktops',
 'Region': 'West',
 'Year': 2024,
 'Quarter': 'Q1',
 'TotalSales': 125000.00,
 'AvgQuantity': 0.40
 },
 # ...
]
```

Aplicamos esto:



```
final_results.sort(
 key=lambda x: x['TotalSales'],
 reverse=True
)
```

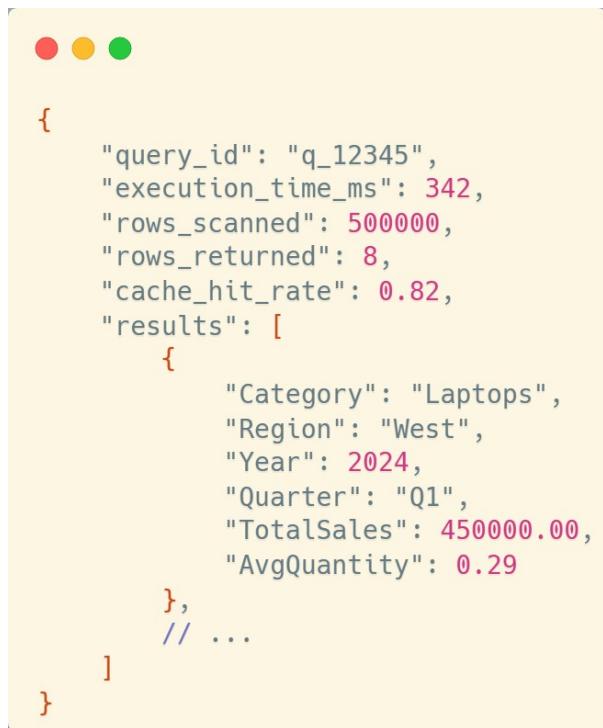
Resultado ordenado:

```
python
[
 {Category: 'Laptops', ..., TotalSales: 450000},
 {Category: 'Laptops', ..., TotalSales: 380000},
 {Category: 'Desktops', ..., TotalSales: 125000},
 # ...
]
```

- **Algoritmo:** QuickSort / MergeSort
- **Complejidad:**  $O(n \log n)$  donde  $n=8$
- **Tiempo:** <1 ms (n muy pequeño)
- **Location:** RAM

## 2.5.5.2.1.7: FASE 7: RESULT FORMATTING (Formato de Salida)

Output format: JSON / XML / Table



```
{
 "query_id": "q_12345",
 "execution_time_ms": 342,
 "rows_scanned": 500000,
 "rows_returned": 8,
 "cache_hit_rate": 0.82,
 "results": [
 {
 "Category": "Laptops",
 "Region": "West",
 "Year": 2024,
 "Quarter": "Q1",
 "TotalSales": 450000.00,
 "AvgQuantity": 0.29
 },
 // ...
]
}
```

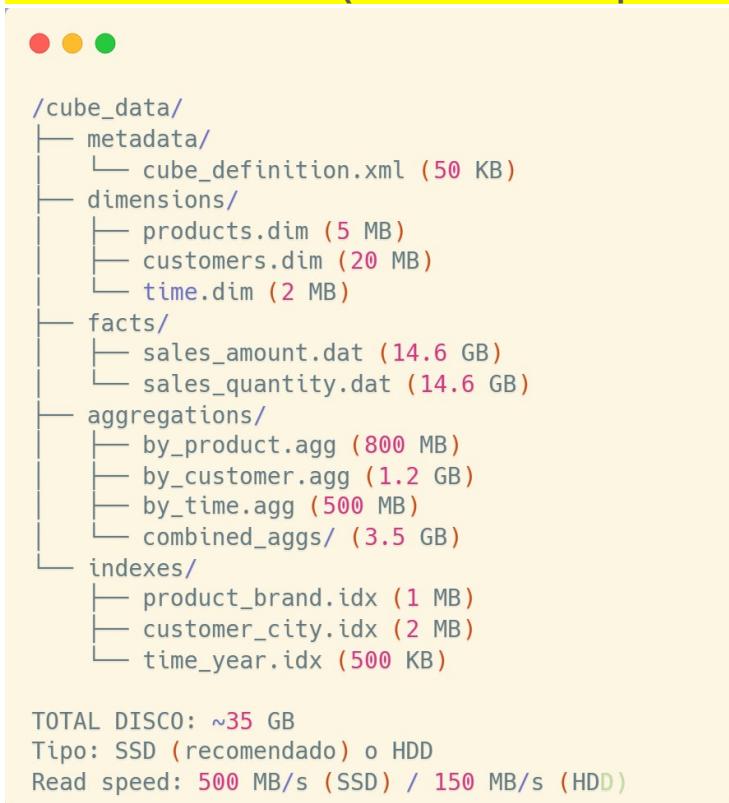
- **Tiempo:** ~2-5 ms
- **Location:** RAM → Network

### 2.5.5.3: 3. MEMORIA vs DISCO: Detalle Completo

Arquitectura de Almacenamiento en MOLAP

#### NIVELES DE ALMACENAMIENTO

##### 2.5.5.3.1: lvl 1: DISCO (Almacenamiento permanente)



##### 2.5.5.3.2: lvl 2: MEMORIA (Cache y datos activados)

#### RAM ALLOCATION:

##### 1. MOLAP Server Process:

- Dimension Cache (1 GB) → Hot dimensions en RAM
- Facts Cache (2 GB) → Bloques recientemente usados
- Aggregation Cache (1 GB) → Aggs frecuentes
- Query Workspace (500 MB) → Resultados intermedios
- Index Cache (500 MB) → B-trees en memoria

##### 2. OS File System Cache:

- Automático (2-4 GB típico)

TOTAL RAM: 5-8 GB típico (para cubo de 35 GB en disco)

##### 2.5.5.3.3: lvl3: CPU (procesamiento)

- L1 Cache: 32-64 KB (por core)
- L2 Cache: 256-512 KB (por core)
- L3 Cache: 8-32 MB (compartido)
- Registers: Datos inmediatos

#### **2.5.5.3.4: Flujo de Datos -> Memoria -> CPU**

**LECTURA DE FACTS: Ejemplo Detallado**

**Query necesita: 9,709,584 celdas**

#### **PASO 1: Determinar ubicación**

Cache manager verifica:

##### **- ¿Está en Facts Cache (RAM)?**

- Si (hit): Leer de RAM
  - Latency: 100 ns
  - Throughput (rendimiento): 20 GB/s
  - Si (hit): Leer de RAM
- No (miss): leer de disco
  - Latency: 5-10 ms (SSD)
  - Throughput: 500 MB/s

#### **PASO 2: Cache miss -> Lectura de disco**

##### **1. Determinar bloques necesarios**

- Archivo: sales\_amount.dat (14.6 GB)
- Block size: 64 KB (típico)
- Total blocks: 228,125

##### **2. Calcular qué bloques leer**

- Offsets needed: [100, 101, ..., 9M]
- Blocks needed: [1, 2, 45, 78, ...]
- (no todos los offsets están en el mismo bloque)

##### **3. Batch I/O request**

- OS lee múltiples bloques
- read\_blocks([1, 2, 45, 78, ...])

##### **4. Flujo de datos**

SSD → OS Cache → MOLAP Cache

##### **5. Datos ahora en RAM para próximas queries (warm cache)**

## PASO 3: Procesamiento en memoria

Datos en RAM:

- Dimension stores (27 MB)
- **Facts block (155 MB para query)**
- Working memory (20 MB)
- **Total RAM usada: ~200 MB para esta query**

CPU opera sobre:

- Datos cargados en cache L3
- Vectorización (SIMD) si es posible
- Multi-threading (4-8 cores)

---

## TIMING BREAKDOWN (IMPORTANTE)

Primera ejecución (cold cache):

- Disco read: 300 ms
- Cache load: 50 ms
- Processing: 100 ms
- **TOTAL: ~450 ms**

Segunda ejecución (warm cache):

- RAM read: 10 ms
- Processing: 100 ms
- **TOTAL: ~110 ms**

Tercera+ ejecución (hot cache):

- L3 cache hit: 1 ms
- Processing: 100 ms
- **TOTAL: ~101 ms**

---

## Políticas de Cache (Qué Se Guarda en Memoria)

### 1. DIMENSION STORES

**Politica:** Casi siempre en RAM

¿Por qué?

- Tamaño pequeño (5-50 MB típico)
- Acceso MUY frecuente
- Lookup O(1) crítico

Estrategia:

- Cargar al inicio del servidor
- Mantener en RAM permanentemente
- Solo refresh cuando se actualiza

## **2. AGGREGATIONS**

**Politica:** Cache selectivo

**Estrategia:**

- Cache aggs mas usadas (LRU)
- Típicamente 20-30% de aggs en RAM
- Resto en disco, carga bajo demanda

**Prioridad:**

1. Grand totals (siempre en RAM)
  2. Aggs de 1 dimension (cache alto)
  3. Aggs combinadas (cache medio)
- 

## **3. FACTS ARRAYS**

**Politica:** Cache LRU (Least Recently Used)

**Estrategia:**

- Solo bloques recientemente usados
- Tipicamente 5-10% en RAM
- **Eviction cuando RAM llena**

**Algorimo:**

1. Query solicita Bloque X
  2. Si en cache -> return
  3. Si no -> cargar desde disco
  4. Si cache llena -> evict LRU
  5. Agregar bloque X a cache
- 

## **4. INDEXES**

**Politica:** Cache agresivo

- B-tress cargados en RAM
- Tamaño pequeño (5-20 MB)
- Criticos para performance

## 2.5.5.3: 4: DATA WAREHOUSE vs DATA MARTS

### ¿Afecta la Arquitectura MOLAP?

Resp: NO AFECTA LA ESTRUCTURA INTERNA

MOLAP no distingue entre DWH y Data Marts

La estructura interna es IDENTICA:

- Dimension Stores
- Facts Arrays
- Aggregations
- Indexes

### Lo que cambia es el SCOPE (alcance)

#### Data Warehouse MOLAP (Corporativo)

Características:

- Todas las áreas de negocio
- Multiples subject áreas
- Datos históricos completos (5-10 años)
- Alta granularidad

Ejemplo - Cubo Corporativo:

● ● ●

corporate\_cube.mdb

Dimensiones:

- Products: 50,000 items
- Customers: 1,000,000 clientes
- Geography: 10,000 locations
- Time: 3,650 days (10 años)
- Channels: 20 canales
- Promotions: 5,000 promociones

Celdas totales: 50K × 1M × 10K × 3650 × 20 × 5K  
= números MASIVOS

Tamaño: 500 GB - 5 TB  
Usuarios: Toda la empresa (1000+)  
Queries: 24/7, muy variados

Estructura MOLAP:

● ● ●

/corporate\_cube/  
└── dimensions/ (500 MB)  
└── facts/ (300 GB)  
└── aggregations/ (150 GB)  
└── indexes/ (50 GB)

MISMA estructura que data mart  
Solo DIFERENTE en tamaño/alcance

## Data Mart MOLAP (Departamental):

### Características:

- Un área específica (ej: ventas)
- Un departamento
- Datos relevantes (2-3 años)
- Granularidad adecuada

### Ejemplo – Data Mart de Ventas:

sales\_mart\_cube.mdb

Dimensiones:

- Products: 1,000 productos activos
- Customers: 50,000 clientes activos
- Stores: 200 tiendas
- Time: 1,095 días (3 años)

Celdas totales:  $1K \times 50K \times 200 \times 1095$   
= 10,950,000,000  
= 10.9 mil millones

Tamaño: 5-50 GB  
Usuarios: Dept de ventas (50-200)  
Queries: Focalizados en ventas

### Estructura MOLAP:

```
/sales_mart_cube/
├── dimensions/ (50 MB)
├── facts/ (20 GB)
├── aggregations/ (10 GB)
└── indexes/ (500 MB)
```

MISMA estructura que DWH  
Sólo para un DEPARTAMENTO y ENFOCADO

## Comparación: DWH vs Data Mart MOLAP

| Aspecto                            | DWH Corporate                                     | Data Mart                                         |
|------------------------------------|---------------------------------------------------|---------------------------------------------------|
| <b>ESTRUCTURA MOLAP (IDÉNTICA)</b> |                                                   |                                                   |
| Dimensions                         | <input checked="" type="checkbox"/> Sí            | <input checked="" type="checkbox"/> Sí            |
| Facts                              | <input checked="" type="checkbox"/> Sí            | <input checked="" type="checkbox"/> Sí            |
| Aggregations                       | <input checked="" type="checkbox"/> Sí            | <input checked="" type="checkbox"/> Sí            |
| Indexes                            | <input checked="" type="checkbox"/> Sí            | <input checked="" type="checkbox"/> Sí            |
| Offset calc                        | <input checked="" type="checkbox"/> Misma fórmula | <input checked="" type="checkbox"/> Misma fórmula |
| <b>DIFERENCIAS (SCOPE)</b>         |                                                   |                                                   |
| Alcance                            | Toda empresa                                      | 1 departamento                                    |
| Dimensiones                        | 10-20 dims                                        | 4-8 dims                                          |
| Cardinalidad                       | Millones                                          | Miles                                             |
| Tamaño                             | 500GB - 5TB                                       | 5-50 GB                                           |
| Usuarios                           | 1000+                                             | 50-200                                            |
| Complejidad                        | Alta                                              | Media                                             |
| Velocidad                          | Más lenta                                         | Más rápida                                        |

### Conclusión:

- La estructura MOLAP es EXACTAMENTE igual
- DWH vs Data Mart solo cambia TAMAÑO y ALCANCE
- El motor MOLAP trata ambos IDÉNTICAMENTE

## **2.5.5.4: 5. TIME() en MOLAP**

**Time NO es una Funcion, es una Dimension Especial**

En MOLAP, "Time" se refiere a:

### **1. TIME DIMENSION STORE**

- una dimensión como cualquier otra, pero con características especiales

### **2. NO es una función** como SQL's NOW()

### **3. ES una dimensión** con jerarquías temporales

---

#### **Estructura de Time Dimension (Detallada)**

**Archivo:** /cube\_data/dimensions/time.dim

**Granularidad base:** DIA

**Member 0:** (2024-01-01)

**Identificadas:**

- DateKey: 20240101 (int)
- FullDate: 2024-01-01 (date)

**Jerarquia Day -> Month -> Quarter -> Year:**

- Day: 1
- DayOfWeek: 1 (Monday)
- DayName: "Monday"
- DayOfMonth: 1
- DayOfYear: 1
- WeekOfYear: 1
- Month: 1
- MonthName: "January"
- MonthShort: "Jan"
- Quarter: 1
- QuarterName: "Q1"
- Year: 2024

**Jejarquia Fiscal (opcional):**

- FiscalDay: 93
- FiscalWeek: 14
- FiscalMonth: 4
- FiscalQuarter: 2
- FiscalYear: 2024

**Flags booleanos:**

- IsWeekend: false
- IsHoliday: false
- IsWorkingDay: true
- IsLastDayOfMonth: false
- IsLastDayOfQuarter: false
- IsLastDayOfYear: false

**Periodos relativos:**

- DaysSinceEpoch: 19723
- WeeksSinceEpoch: 2817

**Periodos de comparación:**

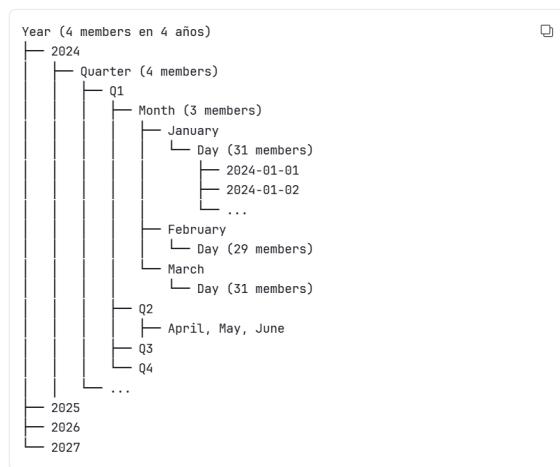
- YearAgo: 20230101
- MonthAgo: 20231201
- QuarterAgo: 20231001

**Total Members:** 3,650 (10 años × 365 días) **Tamaño:** ~2-5 MB

---

## Jerarquías Temporales: Visualización

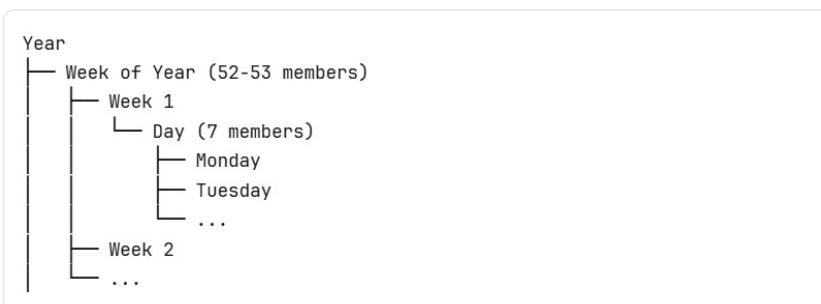
### JERARQUÍA 1: Calendar (predeterminada)



### JERARQUÍA 2: Fiscal (opcional)



### JERARQUÍA 3: Week (opcional)



## 2.5.5.4.1: Time Intelligence Functions en MOLAP

Disponibles en MDX/DAX:

## 1. COMPARACIONES TEMPORALES

### ParallelPeriod()

- Compara con periodo equivalente
- Ejemplo: Este mes vs mismo mes año anterior

```
mdx
```



```
ParallelPeriod(
 [Time].[Calendar].[Month],
 1,
 [Time].[Calendar].CurrentMember
)
```

Resultado: Mes actual -> Mes año pasado

---

## 2. ACUMULADOS (YTD, QTD, MTD)

### PeriodsToDate()

- Suma desde inicio del periodo

Year-to-Date:

```
mdx

Sum(
 PeriodsToDate([Time].[Year], [Time].CurrentMember),
 [Measures].[Sales]
)
```

Quarter-to-Date:

```
mdx

Sum(
 PeriodsToDate([Time].[Quarter], [Time].CurrentMember),
 [Measures].[Sales]
)
```

## 3. MOVING AVERAGES

```
mdx
```

```
Avg(
 LastPeriods(30, [Time].CurrentMember),
 [Measures].[Sales]
)
```

Promedio móvil de últimos 30 días

---

#### 4. GROWTH / VARIANCE

```
mdx
```

```
([Time].CurrentMember, [Measures].[Sales])
- ([Time].PrevMember, [Measures].[Sales])
```

Crecimiento periodo a periodo

##### 2.5.5.4.2: Time en Agregaciones: Optimización Especial

Time dimension tiene **AGREGACIONES ESPECIALES**:

## **1. DAILY LEVEL (base):**

- **Archivo:** /facts/sales\_amount.dat
- **Granularidad:** Dia
- **Celdas:** Products x Customers x Days
- **Tamaño:** Grande

## **2. MONTHLY AGGREGATION**

- **Archivo:** /aggregations/by\_month.agg
- **Granularidad:** Mes
- **Celdas:** Products x Customers x Months
- **Tamaño:** ~1/30 del daily
- **Query:** "Ventas por mes" -> instant

## **3. QUARTERLY AGGREGATION**

- **Archivo:** /aggregations/by\_quarter.agg
- **Granularidad:** Trimestre
- **Celdas:** Products x Customers x Quarters
- **Tamaño:** ~1/90 del daily

## **4. YEARLY AGGREGATION**

- **Archivo:** /aggregations/by\_year.agg
- **Granularidad:** Año
- **Celdas:** Products x Customers x Years
- **Tamaño:** ~1/365 del daily
- **Query:** "Ventas por año" → Instantáneo

### **VENTAJA:**

Si query pide datos mensuales, NO suma 30 días. Solo lee el agg mensual (ya pre-calculado).

### **Ejemplo:**

Query: "Ventas totales de Enero 2024"

- **Sin agg:** Sumar 31 días x productos x clientes

-Tiempo: 500 ms

- **Con agg:** Leer 1 valor del agg mensual

-Tiempo: 1 ms

## **2.5.5.5: RESUMEN EJECUTIVO**

### **2.5.5.5.1: Flujo de Query Completa**

## **7 Fases principales:**

### **1. Query Parsing (- 5 ms)**

- Analisis sintactico
- Identificacion de componentes

### **2. Dimension Resolution (-20 ms)**

- Busqueda en dimension stores
- Filtrado por atributos

### **3. Facts Retrieval (-300 ms)**

- Lectura de arrays
- Filtrado por valores

### **4. Attribute Resolution (-75 ms)**

- Enriquecimiento con nombres

### **5. Aggregation (-30 ms)**

- GROUP BY en memoria
- Calculo de métricas

### **6. Sorting (-1 ms)**

- Ordenamiento de resultados

### **7. Formatting (-5 ms)**

- Preparacion de output

## **2.5.5.5.2: Memoria vs Disco (con aclaracion de que es LRU y LFU – Cache Eviction Policies)**

### **SIEMPRE EN MEMORIA:**

- Dimension Stores (5-50 MB)
- Indexes (1-10 MB)
- Metadata (< 1 MB)

### **CACHE SELECTIVO (RAM):**

- Facts: 5-10% (LRU policy)
- Aggregations: 20-30% (LFU policy)

### **SIEMPRE EN DISCO:**

- Facts base completos (GB-TB)
- Todas las aggregations (GB)
- Particiones históricas
- Backups

## **2.5.5.5.2.1: 1/2: LRU (Least Recently Used)**

**Regla:** Elimina el elemento que **no se ha usado por mas tiempo**.

Cache (capacidad 3):



Acceso: A → [A]

Acceso: B → [B, A]

Acceso: C → [C, B, A]

Acceso: A → [A, C, B] (A se mueve al frente)

Acceso: D → [D, A, C] (B se elimina, era el menos reciente)

↑                   ↑

nuevo           eliminado

**Logica:** "Si no lo has usado recientemente, probablemente no lo necesites"

#### 2.5.5.5.2.2: 1/2: LFU (Least Frequently Used)

**Regla:** Elimina el elemento que **se ha usado menos veces**.

Cache (capacidad 3):



Acceso: A → [A(1)]

Acceso: B → [B(1), A(1)]

Acceso: C → [C(1), B(1), A(1)]

Acceso: A → [A(2), C(1), B(1)] (A: contador +1)

Acceso: A → [A(3), C(1), B(1)]

Acceso: D → [D(1), A(3), C(1)] (B eliminado, menor frecuencia)

↑                   ↑

nuevo           eliminado (freq=1)

**Logica:** "Si no lo usas frecuentemente, probablemente no sea importante"

#### 2.5.5.5.2.3: 1/2: Comparación

| Aspecto | LRU            | LFU             |
|---------|----------------|-----------------|
| Elimina | Menos reciente | Menos frecuente |

|            |                                        |                                                  |
|------------|----------------------------------------|--------------------------------------------------|
| Cuenta     | Tiempo desde ultimo uso                | Total de usos                                    |
| Ventaja    | Simple, funciona bien en general       | Mantiene datos populares                         |
| Desventaja | Un acceso antiguo muy usado se elimina | Un acceso nuevo importante tarda en establecerse |

## En MOLAP

### Facts Cache (LRU):

- Bloques accedidos recientemente se quedan
- Buenos para queries repetitivas

### Aggregations Cache (LFU):

- Agregaciones usadas frecuentemente (ej: grand total)
- Se mantienen aunque pasen días sin usarse

## Regla general:

- **LRU:** Datos temporales, queries cambiantes
- **LFU:** Datos populares, queries repetitivas

## En MOLAP

### Facts Cache (LRU):

- Bloques accedidos recientemente se quedan
- Buenos para queries repetitivas

### Aggregations Cache (LFU):

- Agregaciones usadas frecuentemente (ej: grand total)
- Se mantienen aunque pasen días sin usarse

## Regla general:

- **LRU:** Datos temporales, queries cambiantes
- **LFU:** Datos populares, queries repetitivas

## 2.5.5.5.2.3: 2/2: Diferencia LRU vs LFU – Explicación Simple

### La Diferencia Clave

**LRU:** ¿CUÁNDO fue la ultima vez que lo usaste?

**LFU:** ¿CUANTAS VECES lo has usado en total?

### Ejemplo con Películas en Netflix

Tienes espacio para 3 películas descargadas:

Lunes: Descargas "Avengers"

Martes: Descargas "Titanic"

Miércoles: Descargas "Matrix"

Jueves: Ves "Avengers" otra vez

Viernes: Quieres descargar "Inception" (necesitas borrar 1)

### Con LRU (Least Recently Used):

Estado actual:

- Avengers: vista el JUEVES (hace 1 día) ← Más reciente
- Matrix: descargada el MIÉRCOLES (hace 2 días)
- Titanic: descargada el MARTES (hace 3 días) ← Menos reciente

LRU borra: TITANIC (la que tiene más tiempo sin tocar)

### Con LFU (Least Frequently Used):

Estado actual:

- Avengers: vista 2 veces (lunes + jueves) ← Más frecuente
- Matrix: vista 1 vez (miércoles) ← Menos frecuente
- Titanic: vista 1 vez (martes) ← Menos frecuente

LFU borra: MATRIX o TITANIC (las menos vistas)

(empate, puede borrar cualquiera)

### Caso Donde Son DIFERENTES

Histórial de accesos:



Día 1: A  
Día 2: B  
Día 3: C  
Día 4: A (segunda vez)  
Día 5: A (tercera vez)  
Día 6: B (segunda vez)  
Día 30: C (hace mucho no se usa)

Ahora necesitas espacio. ¿Qué borras?

LRU mira el TIEMPO:

- A: última vez día 5 (hace 25 días)
- B: última vez día 6 (hace 24 días) ← Más reciente
- C: última vez día 30 (¡hoy!) ← MUY reciente

LRU borra: A (más tiempo sin usar)

---

LFU mira la FRECUENCIA:

- A: usada 3 veces
- B: usada 2 veces
- C: usada 2 veces (pero solo 1 vez recientemente)

LFU borra: B o C (menos usadas en total)

**Analogía con Ropa**

## LRU (Cuándo):

Tu regla: "La ropa que no he usado recientemente, la guardo"

Camisa favorita: No la usas hace 6 meses → GUARDAR

Camisa nueva: La usaste ayer → DEJAR AFUERA

## LFU (Cuantas veces):

Tu regla: "La ropa que uso poco en total, la guardo" 

Camisa favorita: La has usado 100 veces en tu vida → DEJAR AFUERA

Camisa nueva: Solo la has usado 1 vez → GUARDAR

## Resumen Visual

### LRU: Timeline (línea de tiempo)

---

hace 10 días      hace 5 días      hoy  
A                    B                    C  
↑                    ↑                    ↑  
BORRAR                MANTENER  
(más antiguo)        (más reciente)

### LFU: Contador de usos

---

1 uso      5 usos      10 usos  
A            B            C  
↑            ↑            ↑  
BORRAR        MANTENER  
(menos usado)        (más usado)

## ¿Cuál es Mejor?



LRU es mejor cuando:

- ✓ Los datos "calientes" cambian con el tiempo
- ✓ Queries diferentes cada día
- ✓ Lo reciente es lo importante

Ejemplo: Noticias, feeds de redes sociales

---

LFU es mejor cuando:

- ✓ Ciertos datos son SIEMPRE populares
- ✓ Queries repetitivas constantemente
- ✓ Lo popular es lo importante

Ejemplo: Home page, productos más vendidos

#### [LRU es mejor cuando:

- Los datos "calientes" cambian con el tiempo
- Queries diferentes cada día
- Lo reciente es lo importante

Ejemplo: noticias, feeds de redes sociales.

---

#### LFU es mejor cuando:

- Ciertos datos son SIEMPRE populares
- Queries repetitivas constantemente
- Lo popular es lo importante

Ejemplo: Home page, productos más vendidos

]

#### Diferencia en UNA frase:

- **LRU:** "¿Hace cuánto lo usaste?"
- **LFU:** "¿Cuántas veces lo has usado?"

### **2.5.5.3 Data Warehouse vs Data Marts**

**NO afecta estructura interna**

**Estructura IDENTICA:**

- Dimension Stores
- Facts Arrays
- Aggregations
- Indexes
- Misma formula offset

**Solo cambia SCOPE:**

- DWH: toda empresa, mas grande
- Data Mart: Un departamento, mas pequeño

### **2.5.5.4 Time Dimensión**

**Time es una DIMENSIÓN, no una función**

**Características:**

- Dimension Store especial
- Jerarquias: Day -> Month -> Quarter -> Year
- 35+ atributos por fecha
- Agregaciones especiales por tiempo
- Funciones MDX/DAX: YTD, QTD, ParallelPeriod()

**NO es SQL's NOW()**

Es un catálogo de fechas con metadatos completos

## 2.6: Operacional Data Storage (2do chat)

### ODS (Operational Data Storage)

ODS

✓ Sometimes a little bit confusing

✓ Different understandings / definitions

ODS



Other data sources



Sales data



CRM system

✓ Operational decision making

ETL



ODS

ODS

✓ No need for long history

✓ Needs to be very current or real-time

ODS



Other data sources



Sales data



CRM system

ETL

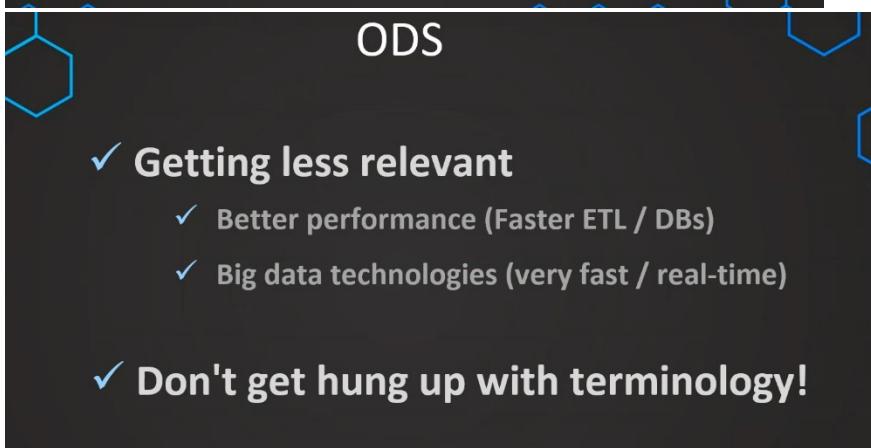
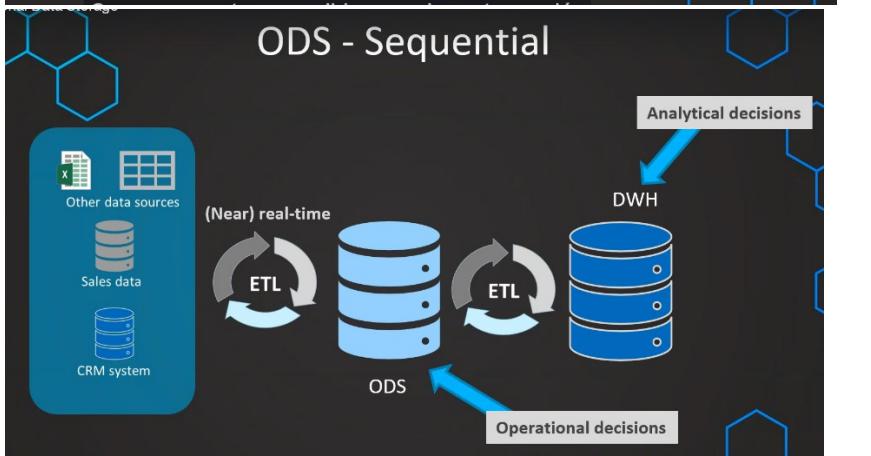
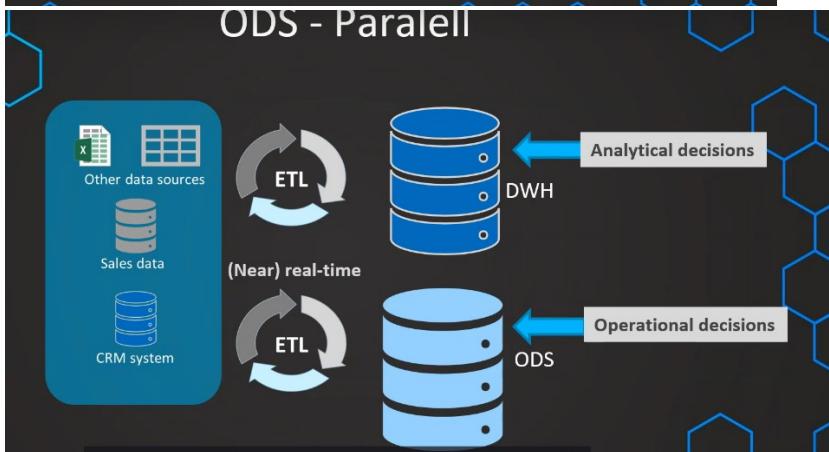
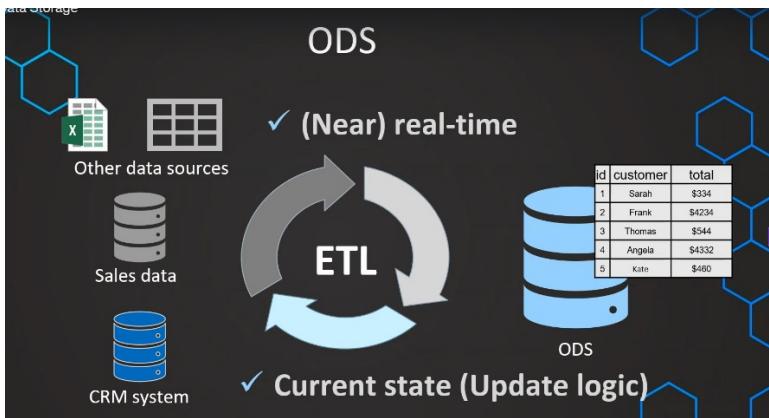
EFT, stock trading

crypto investing

other services

| id | customer | total  |
|----|----------|--------|
| 1  | Sarah    | \$334  |
| 2  | Frank    | \$4234 |
| 3  | Thomas   | \$544  |
| 4  | Angela   | \$4332 |
| 5  | Kate     | \$460  |

ODS



## 2.6.1 ¿Qué es un ODS? (Almacen de Datos Operativos)

Un **ODS (Operational Data Store)** es un tipo de base de datos centralizada diseñada para integrar datos de múltiples sistemas operacionales, proporcionando una vista consolidada y actualizada de los datos operativos de una organización. A diferencia de un Data Warehouse, el ODS se enfoca en datos actuales y en tiempo real para soportar decisiones operativas inmediatas.

### Características Principales

- 1. Datos en tiempo real o casi tiempo real:** EL ODS mantiene información actualizada constantemente desde los sistemas fuente.
- 2. Sin historial extenso:** Solo almacena el estado actual de los datos, no mantiene un historial largo.
- 3. Integración de múltiples fuentes:** Combina datos de diferentes sistemas operacionales en una única ubicación
- 4. Decisiones operativas:** Diseñado para toma de decisiones tácticas del día a día, no estratégicas.
- 5. Actualización constante:** los datos nuevos sobrescriben los datos antiguos (update logic)

## 2.6.2: ODS vs Data Warehouse: Diferencias Clave

| Aspecto                  | ODS                                     | Data Warehouse                           |
|--------------------------|-----------------------------------------|------------------------------------------|
| Propósito                | Decisiones operativas tácticas          | Decisiones estratégicas y analíticas     |
| Tipo de datos            | Estado actual, datos operativos         | Datos históricos, agregados              |
| Actualización            | Tiempo real o casi tiempo real          | Batch (por lotes): diario, semanal       |
| Historial                | Mínimo o inexistente                    | Extenso, datos históricos                |
| Complejidad de consultas | Consultas simples en pequeños volúmenes | Consultas complejas en grandes volúmenes |
| Volatilidad              | Alta - datos cambian constantemente     | Baja - datos son relativamente estáticos |
| Analogía                 | Memoria a corto plazo                   | Memoria a largo plazo                    |
| Costo                    | Más económico (~10% del costo de DWH)   | Más costoso de implementar y mantener    |

## 2.6.3: ¿Por qué Puede Resultar Confuso? + Ejemplo práctico: Empresa de Servicios

### Financieros + Caso de uso: Aprobación de crédito

La distinción entre ODS y Data Warehouse a veces no está clara porque:

- Ambos integran datos de múltiples sistemas operativos
- Ambos utilizan procesos ETL
- Ambos pueden existir en la misma arquitectura de datos
- Las definiciones pueden variar según la fuente o empresa

## Ejemplo Práctico: Empresa de Servicios Financieros

### Escenario

Imagina que trabajas en una empresa de servicios financieros donde los clientes pueden:

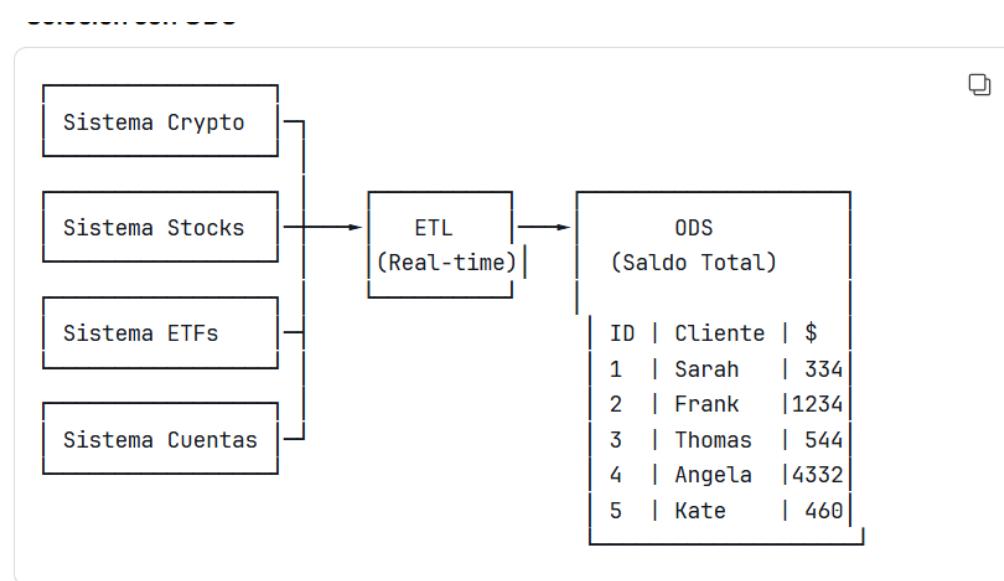
- Invertir en ETFs
- Comerciar acciones
- Invertir en criptomonedas
- Mantener saldo en cuenta

### Problema

Cada servicio está en un sistema diferente:

- Sistema de criptomonedas
- Sistema de trading de acciones
- Sistema de ETFs
- Sistema de cuentas bancarias

### Solución con ODS



## Caso de Uso: Aprobación de Crédito

Cuando un cliente solicita un crédito, necesitas saber **inmediatamente**:

- ¿Cuánto dinero tiene invertido en total?
- ¿Cuál es su saldo actual combinado?
- ¿Puede calificar para el crédito?

El ODS proporciona esta información en **en tiempo** integrando datos de todos los sistemas.

### 2.6.4: Requisitos Técnicos de un ODS

#### 1. No se necesita historial largo

- Solo el estado actual de los datos
- Datos se sobrescriben constantemente

#### 2. actualización en tiempo real o casi tiempo real

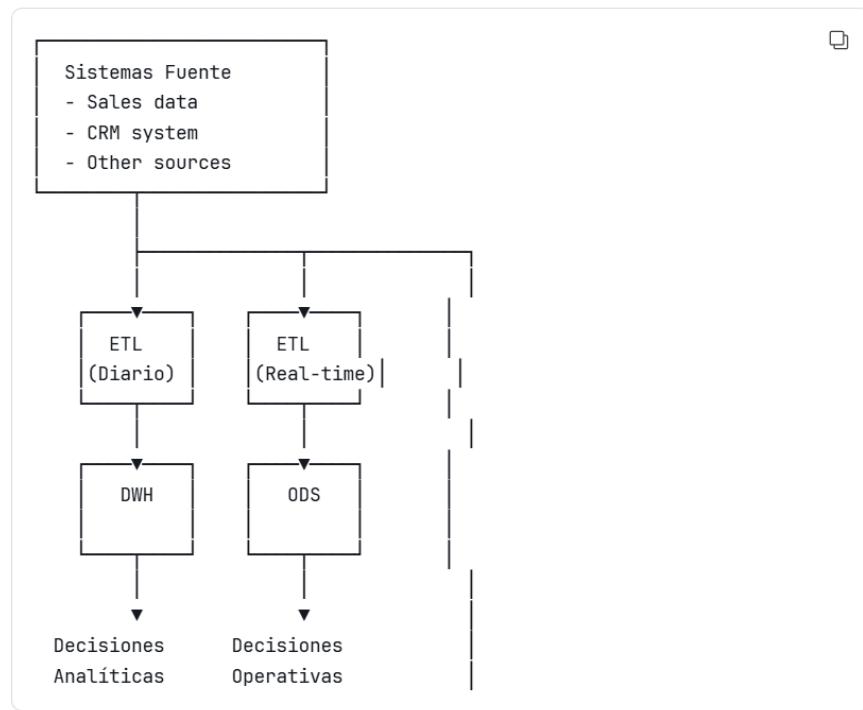
- Los cambios en los sistemas fuente se reflejan inmediatamente
- Critical para decisiones operativas precisas

#### 3. lógica de actualización (Update Logic)

- Los datos NO se agregan como nuevos registros
- Los datos existentes se **reemplazan** o **actualizan**
- Ejemplo: Si Sarah tenía \$334 y ahora tiene \$400, el registro se actualiza a \$400

## 2.6.5: Arquitecturas de Integración con Data Warehouse

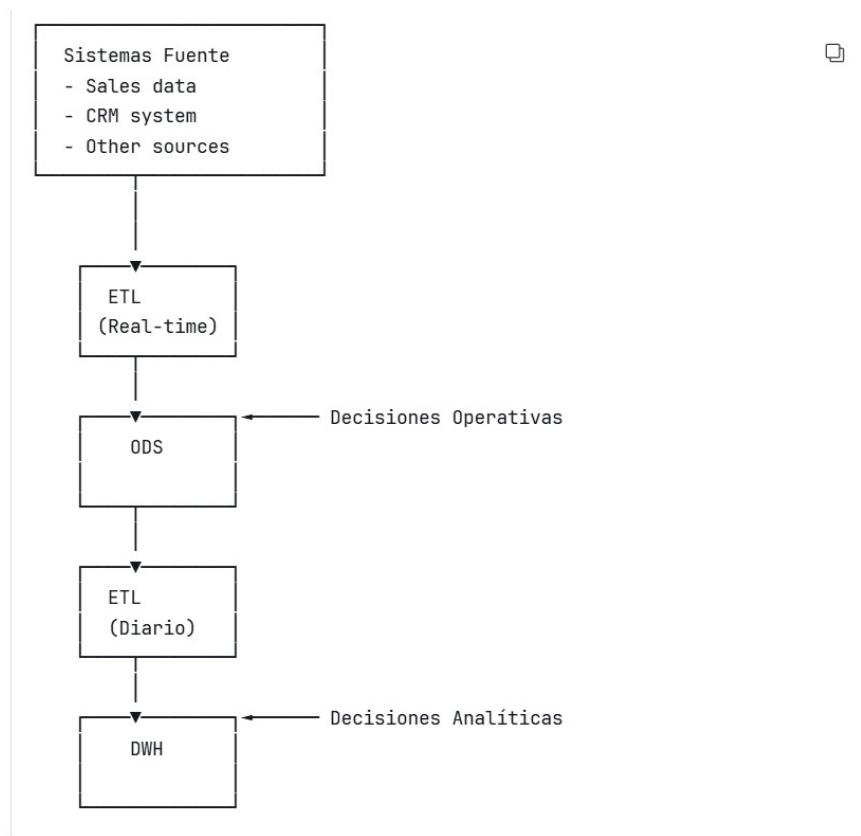
### 2.6.5.1: Arquitectura Paralela



**Ventaja:** Dos ETL independientes con requisitos independientes

**Desventaja:** Duplicación de esfuerzo en la integración de datos

## 2.6.5.2: Arquitectura Secuencial (Mas Común)



### Ventaja:

- El ODS sirve como staging área para el DWH
- Ahorra trabajo de integración
- Los datos ya están integrados en el ODS
- Solo necesitan un ETL adicional del ODS al DWH

## **2.6.6: Ventajas del ODS y Desventajas del ODS**

### **Ventajas del ODS**

- 1. Mas económico:** Hace aproximadamente el 10% del costo de un DWH tradicional
- 2. Consultas simplificadas:** No requiere joins multinivel complejos
- 3. Datos consistentes:** Formatea datos en un solo formato coherente
- 4. Mejora calidad de datos:** Actua como filtro antes del Data Warehouse
- 5. Respuesta inmediata:** Ideal para aplicaciones de cara al cliente
- 6. Minimal transformación:** Los datos mantienen su esquema original

### **Desventajas del ODS**

- 1. No apto para análisis histórico:** Solo mantiene datos actuales
- 2. Consultas limitadas:** No diseñado para consultas analíticas complejas
- 3. Mantenimiento continuo:** Requiere actualización constante
- 4. Complejidad de integración:** Integrar múltiples fuentes puede ser desafiante
- 5. Escalabilidad limitada:** Puede tener problemas con volúmenes masivos de datos

## **2.6.7: Casos de Uso Reales**

### **2.6.7.1 E-commerce: Seguimiento de Pedidos**

**Problema:** Los clientes llaman preguntando: "¿Dónde está mi pedido?"

**Solución ODS:** Integra datos de:

- Plataforma de comercio eléctrico
- Sistema de gestión de almacén
- Transportistas

**Beneficio:** Los representantes de servicio al cliente pueden consultar el ODS y responder inmediatamente con el estado actual del pedido.

### **2.6.7.2: Retail: Dashboard de Ventas en Tiempo Real**

**Problema:** Necesitas saber el rendimiento de ventas en este momento

**Solución ODS:** Dashboard actualizado cada hora mostrando:

- Ventas por tienda
- Disponibilidad de productos
- Tráfico de clientes

### **2.6.7.3: Healthcare: Registros de Pacientes**

**Problema:** Los profesionales de la salud necesitan acceso inmediato a:

- Historial médico del paciente
- Tratamientos actuales
- Planes de cuidado

### **2.6.7.4: Telecomunicaciones: Monitoreo de Red**

**Problema:** Detectar y responder a problemas de red inmediatamente

**Solución ODS:** Integración en tiempo real de:

- Estado de torres celulares
- Tráfico de red
- Incidentes reportados

## 2.6.8: Proceso ETL en ODS + Tendencias Actuales + Recomendación pragmática

### Diferencia con ETL Tradicional

#### ETL Tradicional (hacia DWH):

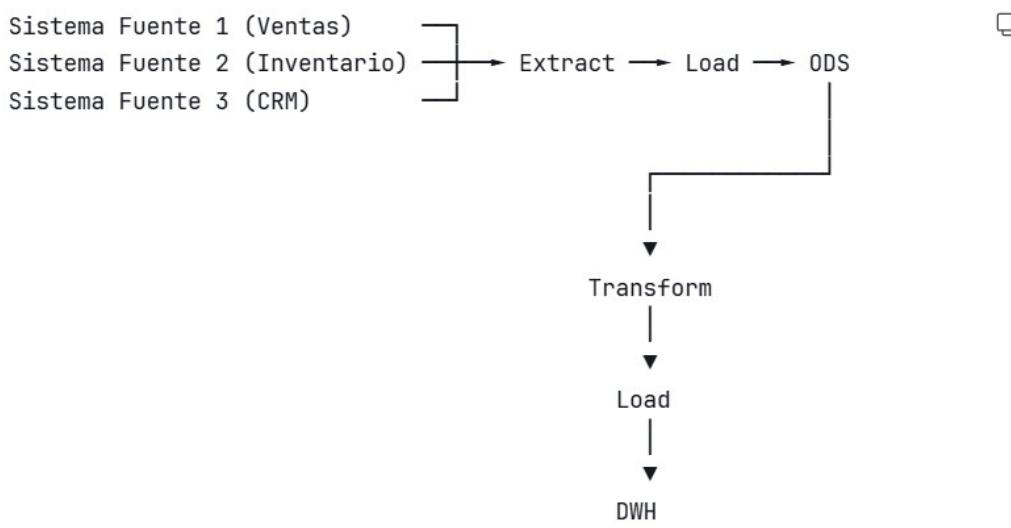
Extract → Transform → Load

#### ETL para ODS:

Extract → Load (sin Transform)

El ODS mantiene los datos en su **formato original**, sin transformaciones complejas. Las transformaciones ocurren **después**, cuando los datos pasan del ODS al Data Warehouse.

### Ejemplo del Flujo



## Tendencias Actuales

### ODS es Cada Vez Menos Relevante

¿Por qué?

- **Mejor rendimiento de hardware:** Podemos cargar datos mucho más rápido ahora
- **Cloud Data Warehouses:** Snowflake, BigQuery, Redshift pueden manejar actualizaciones más frecuentes
- **Nuevas tecnologías de streaming:** Kafka, Apache Flink, Kinesis manejan datos en tiempo real.
- **Data Lakehouses:** Combinan características de Data Lakes y Data Warehouses
- **Change Data Capture (CDC):** Permite captura de cambios en tiempo real sin ODS

## Recomendación Pragmática

**No te obsesiones con la terminología:** Si en tu empresa existe un ODS:

- Utilízalo como fuente para tu ETL hacia el DWH
- Aprovecha el trabajo de integración ya realizado
- Sé pragmático y evalúa si te sirve para tu caso específico

## 2.6.9: Resumen: ¿Cuándo Usar un ODS? + Conceptos Clave para Recordar + Ejemplo Completo: Integración ODS + DWH

### ✓ Usa un ODS cuando:

- Necesitas datos operativos en **tiempo real**
- Requieres decisiones **tácticas inmediatas**
- Tienes múltiples sistemas que necesitas integrar para operaciones diarias
- Necesitas responder preguntas como "¿Cuál es el estado actual de...?"
- **Quieres mejorar la calidad de datos antes de cargar al DWH**
- Tienes restricciones de presupuesto (el ODS es más económico)

### ✗ No uses un ODS cuando:

- Necesitas análisis histórico profundo
- Requiere consultas analíticas complejas
- Necesitas mantener un largo historial de datos
- Tu objetivo es análisis estratégico a largo plazo
- Puedes usar directamente un Cloud Data Warehouse moderno con capacidades de actualización frecuente

---

### Conceptos Clave para Recordar

- **ODS = Decisiones Operativas | DWH = Decisiones Estratégicas**
- **ODS = Tiempo Real | DWH = Histórico**
- **ODS = Estado Actual | DWH = Tendencias a Largo Plazo**
- **ODS = Memoria a Corto Plazo | DWH = Memoria a Largo Plazo**
- **ODS puede servir como staging area para el DWH**

## 2.6.9.1: Ejemplo Completo: Integración ODS + DWH

Empresa: Tienda Online "TechStore"

Sistemas Operacionales:

- Sistema de Pedidos (Order Management)
- Sistema de Inventario (Inventory)
- Sistema de Clientes (CRM)
- Sistema de Pagos (Payment Gateway)

Flujo con ODS Secuencial:

### 2.6.9.1.1: Paso 1: Integración a ODS (Tiempo Real)

```
sql

-- ODS almacena estado actual
CREATE TABLE ods.customer_balance (
 customer_id INT,
 customer_name VARCHAR(100),
 current_balance DECIMAL(10,2),
 last_updated TIMESTAMP
);

-- Cuando Sarah hace una compra, el registro se ACTUALIZA
UPDATE ods.customer_balance
SET current_balance = 250.00,
 last_updated = NOW()
WHERE customer_id = 1;
```

### 2.6.9.1.2: Paso 2: Uso Operativo

- Un representante de servicio puede consultar el ODS para ver el saldo actual de Sarah
- El sistema puede decidir automáticamente si aprobar un crédito

### 2.6.9.1.3: Paso 3: Carga a DWH (Diaria)

```
sql

-- DWH almacena historial
CREATE TABLE dwh.customer_balance_history (
 customer_id INT,
 customer_name VARCHAR(100),
 balance DECIMAL(10,2),
 snapshot_date DATE
);

-- Cada día, se INSERTA un nuevo registro (no se actualiza)
INSERT INTO dwh.customer_balance_history
SELECT customer_id, customer_name, current_balance, CURRENT_DATE
FROM ods.customer_balance;
```

#### 2.6.9.1.4: Paso 4: Análisis Estratégico

```
sql
-- Consulta en DWH: Tendencia de 6 meses
SELECT
 customer_name,
 AVG(balance) as avg_balance,
 MIN(balance) as min_balance,
 MAX(balance) as max_balance
FROM dwh.customer_balance_history
WHERE snapshot_date >= DATE_SUB(CURRENT_DATE, INTERVAL 6 MONTH)
GROUP BY customer_name;
```

#### 2.6.9.2: Conclusión

El **ODS** es una herramienta valiosa en la arquitectura de datos moderna, especialmente cuando necesitas:

- Integración de datos operativos en tiempo real
- Decisiones tácticas inmediatas
- Una staging area eficiente para tu Data Warehouse

Sin embargo, con el avance de tecnologías cloud y streaming, su relevancia ha disminuido. La clave es ser **pragmático**: evalúa tus necesidades específicas y decide si un ODS aporta valor real a tu arquitectura de datos.

**Recuerda:** No te obsesiones con la terminología perfecta. Lo importante es que tu solución de datos resuelva las necesidades operativas y analíticas de tu negocio de manera efectiva y eficiente

## **2.6.10: Agregado: Tecnologías Modernas que Reemplazan al ODS**

### **Por Que el ODS esta Desapareciendo**

Tres tecnologías modernas están reemplazando al ODS:

- 1. Streaming:** Kafka, Apache Flink, Kinesis
- 2. Data Lakehouses:** Delta Lake, Iceberg, Hudi
- 3. CDC:** Captura de cambios en tiempo real

### **2.6.10.1: Tecnologías de Streaming**

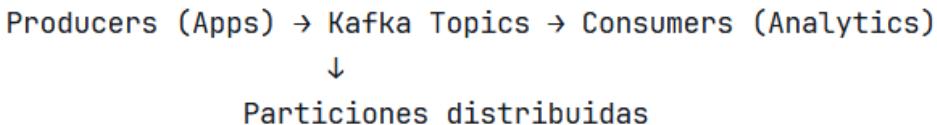
#### **2.6.10.1.1: Apache Kafka**

**Que es:** Plataforma de streaming de eventos distribuida para mover grandes volúmenes de datos en tiempo real.

#### **Características clave:**

- Event log inmutable (append-only)
- Millones de eventos por segundo
- Datos persistentes y escalables

#### **Arquitectura básica:**



#### **Ejemplo práctico - E-commerce:**

```
Sistema Pedidos → Kafka Topic "orders" → [Inventory + Analytics]
```

#### **2.6.10.1.2: Apache Flink**

**Que es:** Motor de procesamiento de streams para transformaciones complejas en tiempo real

#### **Características clave:**

- Procesamiento con estado (stateful)
- Latencia de subsegundos
- Exactly-once semantics

#### **Diferencia con Kafka:**

- Kafka = Almacena y transporta eventos
- Flink = Procesa y transforma eventos

Flujo típico: Kafka (buffer) → Flink (procesamiento) → Destinos

### 2.6.10.1.3: Amazon Kinesis

**Qué es:** Servicio gestionado de AWS para streaming en tiempo real.

**Ventajas:**

- Fully managed (sin infraestructura)
- Auto-scaling
- Integración nativa con AWS

super breve y acá en el mismo chat. aws kinesis hace el trabajo de apache kafka y apache flink?

**Sí, parcialmente:**

**AWS Kinesis tiene 3 componentes:**

**1. Kinesis Data Streams** = hace el trabajo de **Kafka**

- Almacena y transporta eventos en tiempo real

**2. Kinesis Data Analytics** = hace el trabajo de **Flink**

- Procesa streams (basado en Apache Flink internamente)

**3. Kinesis Data Firehose** = Extra

- Carga automática a S3, RedShift, etc.

**Diferencias clave:**

|                | Kafka + Flink   | AWS Kinesis    |
|----------------|-----------------|----------------|
| Gestión        | Tú administras  | AWS administra |
| Costo          | Infraestructura | Pay-per-use    |
| Vendor Lock-in | Open source     | Solo AWS       |
| Flexibilidad   | Total control   | Limitado a AWS |

**Resumen:** Kinesis = Kafka + Flink como **servicio gestionado de AWS**, más fácil pero menos flexible.

### 2.6.10.1.4: Comparación rápida

|                 | Kafka        | Flink       | Kinesis      |
|-----------------|--------------|-------------|--------------|
| <b>Tipo</b>     | Event Store  | Processing  | AWS Managed  |
| <b>Latencia</b> | Milisegundos | Subsegundos | Milisegundos |
| <b>Costo</b>    | Open source  | Open source | Pay-per-use  |

## 2.6.10.2: Data Lakehouses

**Qué es:** Arquitectura que combina Data Lakes (almacenamiento barato) + Data Warehouses (ACID transactions).

**Capas:**

```
Table Format (Metadata) → ACID, Schema, Time Travel
↓
File Formats (Data) → Parquet, ORC, Avro
↓
Object Storage → S3, Azure Blob, GCS
```

### 2.6.10.2.1: Delta Lake (Databricks)

**Características:**

- Transacciones ACID sobre S3
- Time Travel (ver versiones pasadas)
- Integración con Spark

Ejemplo:

```
python

Escribir
df.write.format("delta").save("/path/to/delta")

Time Travel
spark.read.format("delta") \
 .option("timestampAsOf", "2026-02-08") \
 .load("/path/to/delta")
```

### ACID = Garantías de Transacciones en Base de datos

Atomicity (Atomicidad): Todo o nada. Si falla algo, se revierte todo

Consistency (Consistencia): Los datos siempre quedan en estado válido

Isolation (Aislamiento): Las transacciones no se interfieren entre sí.

Durability (Durabilidad): Una vez confirmado, el dato no se pierde

#### Ejemplo práctico:

Transferencia bancaria: \$100 de cuenta A a cuenta B

ACID:

- Atomicity: O se descuenta de A Y se suma a B, o nada
- Consistency: Total siempre = mismo antes y después
- Isolation: Otra transacción no ve el dinero "en el aire"
- Durability: Si dice "exitoso", está guardado permanente

Sin ACID:

- Se descuenta de A pero no llega a B → pierdes \$100

**Por qué importa en Data Warehouses/Lakehouses:** Garantiza que tus datos no se corrompan durante

escrituras concurrentes.

## es como el DB: Transaccion de laravel

¡Exacto! 100

```
php
// Esto ES ACID en Laravel
DB::transaction(function () {
 // A - Atomicity: Todo o nada
 CuentaA::where('id', 1)->decrement('saldo', 100);
 CuentaB::where('id', 2)->increment('saldo', 100);

 // Si algo falla aquí, ROLLBACK automático
 // Si todo OK, COMMIT
});

// C - Consistency: Laravel valida constraints de BD
// I - Isolation: Cada transacción ve datos consistentes
// D - Durability: Una vez commit, queda guardado
```

## Sin DB::transaction:

```
php
// ✗ NO ACID - Peligroso
CuentaA::where('id', 1)->decrement('saldo', 100); // ✓ OK
// ✗ Error aquí
CuentaB::where('id', 2)->increment('saldo', 100); // ✗ Nunca se ejecuta
// Resultado: Se perdieron $100
```

## En Data Lakehouses (Delta/Iceberg/Hudi):

- Hacen lo mismo que DB::transaction() pero en archivos de S3
- Antes era imposible tener ACID en Data Lakes
- Ahora Delta Lake te da ACID como si fuera PostgreSQL

### 2.6.10.2.2: Apache Iceberg (Netflix)

Características:

- Schema evolution sin reescribir datos
- Partition evolution
- Multi-engine (Spark, Flink, Trino, Hive)

**Ventaja clave:** Cambiar particionamiento sin reescribir nada

### 2.6.10.2.2.1: ¿Qué es Particionar? + El problema que Resuelve Iceberg

Es dividir tus datos en carpetas para que las consultas sean mas rápidas

**Ejemplo Real: Tienda Online**

**Sin particionar (todo en un solo archivo):**

```
/data/ventas.parquet (10 TB, 1000 millones de registros)
```

Si buscas ventas del 15 de febrero -> tiene que leer **TODO el archivo de 10TB** 🤯

**Con particiones** (dividido en carpetas):

```
/data/
└── year=2026/
 └── month=02/
 ├── day=01/ventas.parquet
 ├── day=02/ventas.parquet
 ...
 └── day=15/ventas.parquet ← Solo lee ESTA carpeta
```

Ahora solo lee la carpeta day=15 → **100 veces más rápido** ⚡

### El Problema que Resuelve Iceberg

**Escenario:**

- Año 2025: Particionaste por **día** (suficiente)
- Año 2026: Ahora tienes MAS datos, necesitas particionar por **hora**

Sin Iceberg:

✗ Tienes que:

1. Leer TODOS los datos (semanas)
2. Reescribirlos en nueva estructura (semanas)
3. Borrar los viejos

= Proceso de 1 mes, sistema parado

**Con Iceberg:**

- Cambias el esquema de particiones
  - Datos viejos: siguen en /day=15/
  - Datos nuevos: van a /day=15/hour=10/
  - Todo sigue funcionando
- = Proceso de 5 minutos



## Analogía Final

**Partitionar** = Organizar ropa en tu closet:

- Sin partitionar: Todo mezclado en una caja
- Con particiones: Camisas en un cajón, pantalones en otro

Iceberg: Te deja cambiar de "organizado por tipo" a "organizado por color" sin tener que doblar toda la ropa de nuevo.

eso es lo que no entiendo. como eso que cambiar el esquema de particiones.

como eso de datos viejos, no entiendo

## Situación Inicial

Tienes datos de **enero hasta febrero 14**:

```
/data/
└── day=2026-01-01/ventas.parquet
└── day=2026-01-02/ventas.parquet
└── day=2026-01-03/ventas.parquet
...
└── day=2026-02-14/ventas.parquet ← Último dato viejo
```



**Estos son los "datos viejos"** - ya están escritos en disco desde hace semanas.

## Cambias el Esquema (15 de febrero)

Decides: "A partir de HOY quiero particionar por hora, no por día"

## Sin Iceberg ✗:

Tienes que:

1. Leer day=2026-01-01/ventas.parquet
2. Dividirlo en day=2026-01-01/hour=00/, hour=01/, hour=02/...
3. Hacer eso para TODOS los 45 días anteriores
4. Reescribir terabytes de datos

= Semanas de trabajo

## Con Iceberg ✓:

python

```
1. Solo ejecutas este comando (5 minutos)
table.updateSpec()
 .addField(hour("timestamp"))
 .commit()

2. Datos VIEJOS (ene-feb 14): No se tocan
/data/day=2026-01-01/ventas.parquet ← Quedan así
/data/day=2026-02-14/ventas.parquet ← Quedan así

3. Datos NUEVOS (feb 15 en adelante): Usan nuevo esquema
/data/day=2026-02-15/hour=00/ventas.parquet ← Nuevo formato
/data/day=2026-02-15/hour=01/ventas.parquet ← Nuevo formato
/data/day=2026-02-15/hour=10/ventas.parquet ← Nuevo formato
```

## ¿Cómo Funciona Cuando Consultas?

sql

```
SELECT * FROM ventas
WHERE fecha = '2026-01-15' -- Dato viejo
```

👉 Iceberg sabe: "Este dato está en /day=2026-01-15/"

sql

```
SELECT * FROM ventas
WHERE fecha = '2026-02-15' AND hora = 10 -- Dato nuevo
```

👉 Iceberg sabe: "Este dato está en /day=2026-02-15/hour=10/"  
**Iceberg maneja ambos esquemas automáticamente** - tú solo consultas y él encuentra los datos.

Iceberg NO reorganiza datos viejos, solo cambia cómo guardar los nuevos 👍

### 2.6.10.2.2.2: Iceberg (Netflix)

**Problema que resuelve:** Tienes una tabla GIGANTE particionada por día, pero ahora necesitas particionar por hora. Normalmente tendrías que **reescribir toda la tabla** (días/semanas).

## Solucion Iceberg:

```
python

Antes: particionado por día
/data/year=2026/month=02/day=15/

Cambias a particionado por hora SIN reescribir
table.updateSpec()
 .removeField("day")
 .addField(hour("timestamp"))
 .commit()

Ahora: /data/year=2026/month=02/day=15/hour=10/
Los datos viejos siguen ahí, los nuevos usan el nuevo esquema
```

**En términos simples:** Es como cambiar la estructura de carpetas de tu disco duro sin tener que mover todos los archivos.

### 2.6.10.2.3: Apache Hudi (Uber)

#### Características:

- Optimizado para upserts/deletes

- Streaming nativo
- DeltaStreamer para ingestión

Ideal para: Streaming, IoT, GDPR compliance

### 2.6.10.2.3.1: Problema que Resuelve Hudi 1/2

En **Data Lakes tradicionales** (archivos Parquet en S3), **NO puedes hacer UPDATE ni DELETE** fácilmente.

#### Ejemplo Real: Base de Datos de Usuarios

Tienes:

```
/data/users/users.parquet (10 millones de usuarios)
```

**Usuario con id=123 cambia su email:**

**Sin Hudi** ❌

```
python
En base de datos normal (MySQL, PostgreSQL)
UPDATE users SET email = 'nuevo@email.com' WHERE id = 123;
✅ Rápido, solo actualiza 1 registro

En Data Lake sin Hudi
❌ IMPOSIBLE actualizar 1 registro en archivo Parquet
Parquet es INMUTABLE (no se puede editar)

Tienes que:
1. Leer TODO el archivo (10 millones de registros)
2. Encontrar el registro con id=123
3. Cambiar el email
4. REESCRIBIR TODO el archivo completo
5. Borrar el archivo viejo

= Procesar 10 millones de registros para cambiar 1 solo 🚨
```

**Con Hudi** ✓

```
python
Creas un dataframe con SOLO el cambio
df_updates = spark.createDataFrame([
 (123, 'nuevo@email.com')
], ['id', 'email'])

Hudi hace UPSERT eficiente
df_updates.write.format("hudi")
 .option("operation", "upsert") # ← Magia aquí
 .save("/data/users")

Hudi internamente:
- Solo reescribe el archivo pequeño donde está id=123
- El resto de archivos NO se tocan
= Rápido ✨
```

#### Casos de Uso (de tu imagen)

1. **IoT:** Sensor actualiza su estado cada segundo

```
python
```

```
Sensor 456 reporta nueva temperatura
sensor_update.write.format("hudi")
 .option("operation", "upsert")
 .save("/data/sensors")
```

## 2. **GDPR**: Usuario pide borrar sus datos

```
python
```

```
DELETE eficiente
df_delete.write.format("hudi")
 .option("operation", "delete")
 .save("/data/users")
```

## 3. **Streaming**: Datos llegan en tiempo real y se actualizan

```
python
```

```
Stream de pedidos que cambian de estado
"pending" → "shipped" → "delivered"
stream.writeStream.format("hudi")
 .option("operation", "upsert")
 .start("/data/orders")
```

## Resumen

**Hudi** = Hace que archivos Parquet en S3 se comporten como una base de datos MySQL donde puedes hacer **UPDATE** y **DELETE** eficientemente.

Sin Hudi: Actualizar 1 registro = reescribir millones

Con Hudi: Actualizar 1 registro 0 solo reescribe lo necesario

### 2.6.10.2.3.2: Problema que Resuelve Hudi 2/2

**Problema que resuelve:** Tienes millones de registros de usuarios y necesitas **actualizar o borrar** datos constantemente (ej: usuario cambia email, GDPR requiere borrar datos).

#### Solución Hudi:

```
python

Base de datos tradicional
UPDATE users SET email = 'nuevo@email.com' WHERE id = 123;

En Data Lake sin Hudi: ❌ Imposible actualizar
Tendrías que reescribir TODO el archivo Parquet

Con Hudi: ✓ Hace upsert eficiente
df_updates.write.format("hudi")
 .option("operation", "upsert") # ← Actualiza o inserta
 .save("/data/users")
```

#### Casos de uso:

- **IoT:** Sensores envían datos constantemente, necesitas acutalizar estado
- **GDPR:** Usuario pide “borrar mis datos” -> DELETE eficiente
- **Straming:** Datos llegan en tiempo real y se actualizan

#### 2.6.10.2.4: Comparación rápida

|                     | Delta Lake          | Iceberg                | Hudi                |
|---------------------|---------------------|------------------------|---------------------|
| Mejor para          | Uso general + Spark | Multi-engine analytics | Streaming + Upserts |
| Partition Evolution | ✗                   | ✓                      | ✗                   |
| Streaming           | ✓                   | En desarrollo          | ✓ ✓                 |
| Ecosistema          | Databricks          | Netflix, Apple         | Uber                |

#### Cuándo Usar Cada Uno

- **Delta Lake:** Ecosistema DataBricks, uso general
- **Iceberg:** Multi-engine, datasets petabyte, schema flexibility
- **Hudi:** Streaming intensive, muchos upserts/deletes

### **2.6.10.3: Change Data Capture (CDC)**

**Que es:** Captura cambios en bases de datos en tiempo real sin impacto en el sistema fuente.

**Como funciona:**

**Métodos principales:**

#### **1. Log-Based CDC** (Recomendado)

- Lee el transaction log de la BD (WAL, binlog)
- Sin impacto en performance
- Captura INSERT, UPDATE, DELETE

#### **2. Trigger-Based:**

- Triggers en cada tabla
- **✗** Impacta performance

#### **3. Timestamp-Based:**

- Consulta por timestamp
- **✗** No captura DELETEs

### **Arquitectura Log-Based**

Base de Datos → Transaction Log → CDC Tool → Kafka → Destinos  
(Debezium)

### **2.6.10.3.1: Herramientas Populares – Debezium**

#### **Debezium (Open Source):**

- Integración con Kafka
- Soporta: PostgreSQL, MySQL, MongoDB, SQL Server

Ejemplo de evento:

```
json
{
 "before": {"order_id": 123, "status": "pending"},
 "after": {"order_id": 123, "status": "shipped"},
 "op": "u", // update
 "ts_ms": 1708000500000
}
```

#### **Otras herramientas:**

- Striim (Enterprise)
- Oracle GoldenGate
- AWS DMS

#### **Casos de Uso**

- Migración cloud zero-downtime
- Replicación en tiempo real
- Data Lake hydration
- Sincronización de búsqueda
- Cache invalidation

### **2.6.10.3.2: CDC vs ODS**

| Aspecto        | ODS           | CDC               |
|----------------|---------------|-------------------|
| Latencia       | Minutos/Horas | Milisegundos      |
| Impacto fuente | ETL queries   | Sin impacto (log) |
| Complejidad    | Alta          | Media             |
| Costo          | Alto          | Menor             |

#### **2.6.10.4: comparación Final: Arquitecturas**

##### **Tradicional (con ODS)**

Fuentes → ETL → ODS → ETL → DWH

- ✗ Doble ETL
- ✗ Latencia acumulada
- ✗ Costos altos

##### **Moderna (sin ODS)**

Fuentes → CDC → Kafka → Flink → Lakehouse

- ✓ Tiempo real
- ✓ Una fuente de verdad
- ✓ Menor costo

#### **2.6.10.4.1: Data Lakehouse vs Delta Lake**

No son lo mismo – uno es concepto, el otro es tecnología

#### **Data Lakehouse = CONCEPTO/ARQUITECTURA**

Es una **idea/patrón arquitectónico**:

Data Lakehouse = Data Lake + Data Warehouse combinados



Características:

- Almacenamiento barato (como Data Lake)
- ACID transactions (como Data Warehouse)
- Schema enforcement (como Data Warehouse)
- Time travel
- Soporta streaming y batch

Analogía: "Casa inteligente" es un concepto

#### **Delta Lake = TECNOLOGÍA/IMPLEMENTACIÓN**

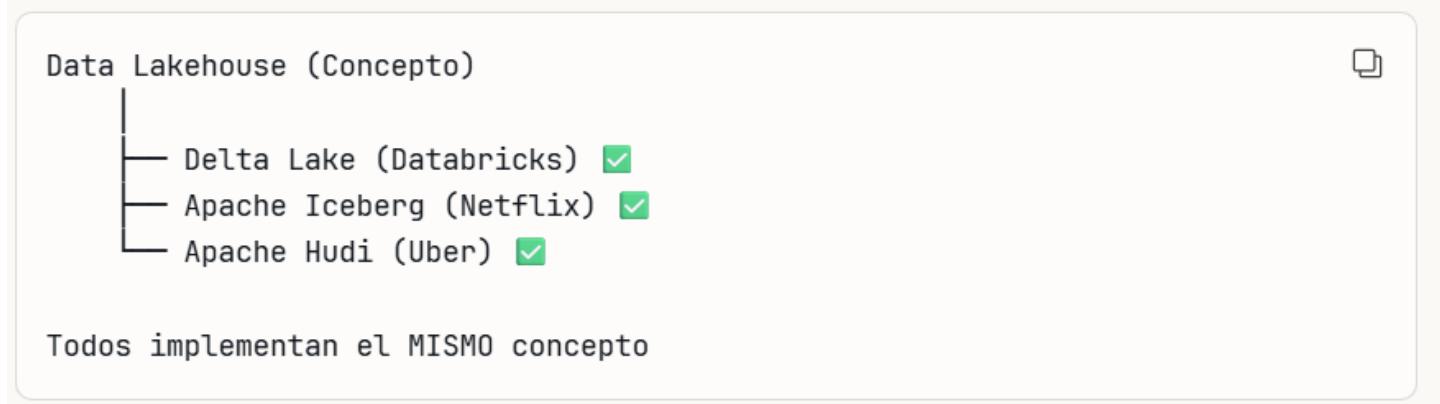
Es una **herramienta específica** que implementa el concepto de Data Lakehouse:

Delta Lake = Software de Databricks

Es UNA forma de construir un Data Lakehouse

Analogia: "Google Nest" es una implemtacion de cada inteligente

## 2.6.10.4.2: Otras Implementaciones de Data Lakehouse



### En Resumen

**Data Lakehouse:** "Quiero combinar Data Lake + Data Warehouse"

**Delta Lake:** "OK, usa esta tecnología para lograrlo"

Es como:

- **Streaming** (concepto) vs **Kafka** (tecnología)
- **Base de datos relacional** (concepto) vs **MySQL** (tecnología)
- **Data Lakehouse** (concepto) vs **Delta Lake** (tecnología)

## 2.6.10.5: Conclusión

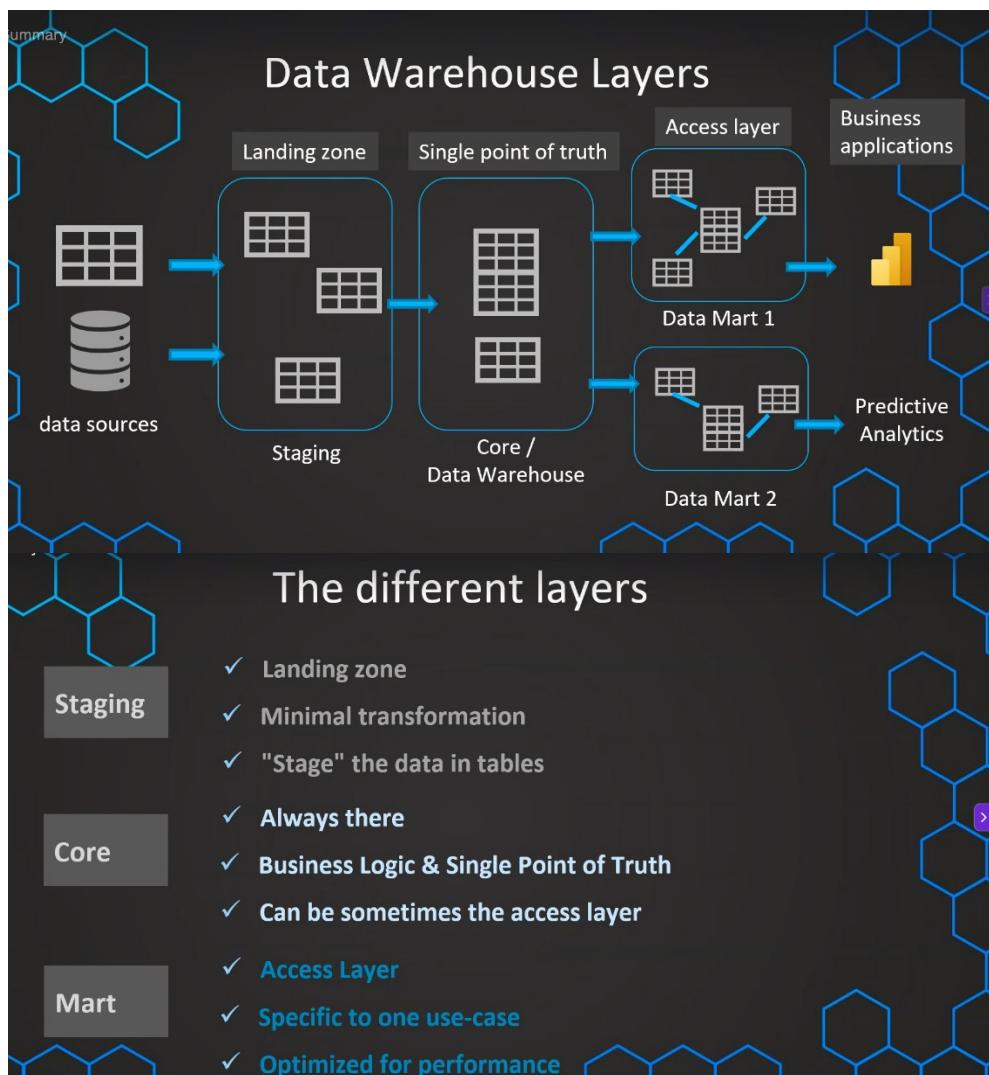
Por qué el ODS desaparece:

1. **Kafka/Flink/Kinesis:** Streaming real, no buffer intermedio
2. **Delta/Iceberg/Hudi:** Operaciones + Analytics en uno
3. **CDC:** Captura más eficiente que ETL

Resultado:

- Arquitecturas mas simples
- Menor latencia (ms vs minutos)
- Costos reducidos
- Mayor escalabilidad

## 2.7: Resumen de todo lo visto



## Resumen de Capas del Data Warehouse

Ahora que hemos aprendido tantas cosas diferentes sobre la arquitectura de un almacén de datos, queremos tomarnos el tiempo para resumir rápidamente lo que hemos aprendido hasta ahora.

## Composición del Data Warehouse

Un almacén de datos, esta es la parte importante, se compone de diferentes capas.

Tenemos nuestras **fuentes de datos** en el principio, y luego usamos nuestra **capa de staging** (puesta en escena) como una especie de zona de aterrizaje donde tenemos todos los datos cargados en tablas en una base de datos.

Y luego tenemos nuestra **capa core** (núcleo). Esta capa está siempre disponible y nos sirve como único punto de verdad. Así que esto significa que esta es la fuente de datos, la única fuente de datos para nuestros data marts, que suelen ser nuestra capa de acceso.

Hemos aprendido que también el core puede, en algunos casos muy simples, ser utilizado como una capa de acceso, pero por lo general queremos construir diferentes data marts con un propósito especial o un caso de uso especial, y luego se optimizan para un rápido rendimiento y acceso a los datos. Y luego se utilizan para distintas aplicaciones o distintos grupos de usuarios.

Esto es lo que hemos aprendido sobre las capas.

## Detalle de las Capas

Vamos a resumir rápidamente lo que hemos aprendido sobre esas diferentes capas con un poco más de detalle.

### Capa de Staging (Puesta en Escena)

Hemos aprendido que esta es la zona de aterrizaje. Así que aquí es donde vamos a cargar primero todos los datos de las diferentes fuentes de datos, y queremos hacer lo menos transformaciones como sea posible. Así que queremos ser aquí lo menos intrusivos posible. Así que básicamente el propósito aquí es poner en escena los datos en nuestra base de datos y tablas.

### Capa Core (Núcleo)

Luego, tenemos la capa core. Por lo general, esta no es nuestra capa de acceso, pero como se ha mencionado, en algunos casos, puede serlo. Pero esto sería más bien una excepción.

Por lo tanto, siempre hay un core que contiene la lógica empresarial. Así que aquí hemos hecho todas las transformaciones de datos. Los datos se transforman y modelan de forma dimensional y esto sirve como **único punto de verdad**.

Esto significa que aquí todos los datos están disponibles en un único almacenamiento. Y ahora los diferentes data marts pueden venir y utilizar esta capa core como fuente de datos. Esto es lo que entendemos por punto único de verdad. Contiene todos nuestros datos en el estado correcto y en el estado transformado. Y todos los diferentes data marts pueden extraer los datos de este core.

Así que, como se ha mencionado, a veces este core también se puede utilizar como capa de acceso. Si tenemos un único caso de uso y si la lógica de nuestro almacén de datos es muy sencilla, esto puede ser posible. Pero es bastante infrecuente.

## **Data Marts (Capa de Acceso)**

Normalmente, también tenemos un data mart que se añade al core. Así que el core sirve como fuente de datos, y luego este data mart es la capa de acceso final que se hace para un caso de uso específico.

Esto puede tener sentido porque queremos que nuestro data mart sea lo menos complejo posible por razones de rendimiento y de facilidad de uso.

Si queremos mejorar el rendimiento, podemos utilizar una base de datos multidimensional o bases de datos en memoria para aumentar aún más el rendimiento.

Así pues, un data mart es nuestra capa de acceso y, a menudo, también está optimizado para el rendimiento si nuestra base de datos relacional no satisface nuestras necesidades de rendimiento.

## **Conclusión**

Esto es lo que hemos aprendido sobre las distintas capas de nuestro almacén de datos.

En la próxima lección, queremos aprender a modelar los datos. Hemos aprendido que modelamos los datos de forma dimensional, en un esquema de estrella, y en la siguiente sección queremos aprender qué significa eso exactamente y cómo se ve en la práctica.