Taller C1: Métricas de Calidad en los Datos para Almacenamiento en Big Data y NoSQL

1. Introducción

En entornos de Big Data y NoSQL, donde los datos son masivos, heterogéneos y a menudo no estructurados, la calidad de los datos es crucial para garantizar análisis útiles y decisiones confiables. Las métricas de calidad evalúan cómo los datos se transforman, cómo se convierten en información significativa y qué tan bien cumplen con estándares de excelencia. Este taller se divide en tres secciones: **Transformación del dato**, **Transformación de la información** y **Calidad del dato**, con ejemplos aplicados a NoSQL.

2. Transformación del Dato

Estas métricas describen cómo los datos crudos se procesan para hacerlos útiles en almacenamiento y análisis.

a) Contextualización

Definición: Agregar metadatos o contexto para que el dato sea comprensible. **Relevancia en Big Data/NoSQL:** En NoSQL, los datos no tienen un esquema fijo, por lo que el contexto es esencial para interpretar documentos JSON. **Ejemplo:** Un sensor IoT genera datos numéricos:

- Dato crudo: {"valor": 25}
- Contextualizado: {"valor": 25, "unidad": "°C", "ubicacion": "Madrid", "fecha": "2025-03-25"}
- Impacto: Sin contexto, el dato es inútil; con él, se puede analizar el clima.

b) Categorización

Definición: Clasificar datos en grupos o tipos. **Relevancia:** Facilita consultas y agregaciones en sistemas NoSQL con grandes volúmenes. **Ejemplo:** Clasificar transacciones:

- Crudo: {"monto": 100, "fecha": "2025-03-25"}
- Categorizado: {"monto": 100, "fecha": "2025-03-25", "categoria": "venta"}
- Impacto: Permite filtrar rápidamente por tipo en Big Data.

c) Cálculo

Definición: Derivar nuevos valores a partir de datos existentes. **Relevancia:** Reduce redundancia al precalcular valores en NoSQL. **Ejemplo:** Calcular el total de un pedido:

- Crudo: {"items": [{"precio": 20}, {"precio": 30}]}
- Calculado: {"items": [{"precio": 20}, {"precio": 30}], "total": 50}
- Impacto: Evita recalcular en cada consulta, optimizando el almacenamiento.

d) Corrección

Definición: Detectar y reparar errores en los datos. **Relevancia:** En Big Data, los datos de fuentes diversas suelen tener inconsistencias. **Ejemplo:** Corregir un error de formato:

- Crudo: {"edad": "veinticinco"}
- Corregido: {"edad": 25}
- Impacto: Datos uniformes mejoran la calidad del análisis.

e) Agregación

Definición: Combinar datos para resumirlos. **Relevancia:** Reduce el volumen de datos almacenados en NoSQL. **Ejemplo:** Resumir ventas diarias:

- Crudo: [{"monto": 50, "fecha": "2025-03-25"}, {"monto": 30, "fecha": "2025-03-25"}]
- Agregado: {"fecha": "2025-03-25", "ventas_totales": 80}
- Impacto: Menor tamaño y mayor eficiencia en consultas.

3. Transformación de la Información

Estas métricas evalúan cómo los datos se convierten en información accionable.

a) Comparación

Definición: Relacionar datos para identificar patrones o diferencias. **Relevancia:** En NoSQL, permite cruzar datos distribuidos. **Ejemplo:** Comparar ventas entre regiones:

- Datos: {"region": "Madrid", "ventas": 500}, {"region": "Barcelona", "ventas": 600}
- Información: "Madrid tiene un 20% menos ventas que Barcelona"
- Impacto: Facilita decisiones basadas en tendencias.

b) Repercusión

Definición: Evaluar el impacto de un dato en el sistema. **Relevancia:** En Big Data, un dato erróneo puede afectar modelos masivos. **Ejemplo:** Un dato atípico:

- Crudo: {"ventas": [50, 60, 1000000]}
- Repercusión: "El valor 1000000 distorsiona la media"
- Impacto: Identificar outliers mejora la calidad.

c) Conexión

Definición: Vincular datos para crear relaciones significativas. **Relevancia:** En NoSQL, las relaciones suelen ser implícitas (embebidas o referenciadas). **Ejemplo:** Vincular usuario y pedido:

- Datos: {"usuario_id": "U001", "nombre": "Ana"}, {"pedido_id": "P001", "usuario_id": "U001"}
- Conexión: "Ana realizó el pedido P001"
- Impacto: Habilita análisis relacional sin esquemas rígidos.

d) Conversación

Definición: Traducir datos a un formato narrativo o comunicable. **Relevancia:** Convierte datos técnicos en insights para humanos. **Ejemplo:** Datos de tráfico:

- Crudo: {"visitas": 1000, "fecha": "2025-03-25"}
- Conversación: "El 25 de marzo, el sitio tuvo 1000 visitas"
- Impacto: Facilita la comunicación con stakeholders.

4. Calidad del Dato

Estas métricas miden la excelencia intrínseca de los datos almacenados.

a) Completitud

Definición: Grado en que los datos tienen todos los atributos necesarios. **Relevancia:** En NoSQL, la falta de esquema puede generar datos incompletos. **Ejemplo:**

- Incompleto: {"nombre": "Ana"}
- Completo: {"nombre": "Ana", "edad": 28, "correo": "ana@example.com"}
- Impacto en Big Data: Datos incompletos dificultan análisis predictivos.

b) Credibilidad

Definición: Confianza en la fuente o veracidad del dato. **Relevancia:** En Big Data, las fuentes heterogéneas pueden ser poco fiables. **Ejemplo:**

- No creíble: {"temperatura": 1000, "unidad": "°C"} (imposible en la Tierra)
- Creíble: {"temperatura": 25, "unidad": "°C", "fuente": "sensor_calibrado"}
- Impacto: Datos no creíbles distorsionan resultados.

c) Precisión

Definición: Exactitud de los datos respecto a la realidad. **Relevancia:** Errores pequeños en NoSQL se amplifican en grandes volúmenes. **Ejemplo:**

- Impreciso: {"distancia": 5.2, "unidad": "km"} (real: 5.18 km)
- Preciso: {"distancia": 5.18, "unidad": "km"}
- Impacto: Mejora modelos analíticos.

d) Consistencia

Definición: Uniformidad de los datos a través del sistema. **Relevancia:** En NoSQL, sin esquema fijo, las inconsistencias son comunes. **Ejemplo:**

- Inconsistente: {"edad": 25}, {"edad": "veinticinco"}
- Consistente: {"edad": 25}, {"edad": 25}
- Impacto: Reduce errores en agregaciones.

e) Interpretabilidad

Definición: Facilidad para entender los datos. **Relevancia:** En Big Data, datos mal estructurados son inútiles. **Ejemplo:**

- Poco interpretable: {"val": 25}
- Interpretable: {"temperatura": 25, "unidad": "°C"}
- Impacto: Mejora la usabilidad en análisis.

5. Ejemplo Completo en NoSQL (JSON)

Consideremos un sistema de monitoreo de tráfico en Big Data almacenado en JSON:

Dato crudo:

```
{"id": "S001", "val": 500, "t": "2025-03-25"}
```

Transformación del dato:

- Contextualización: {"id": "S001", "valor": 500, "tipo": "vehículos", "fecha": "2025-03-25", "ubicacion": "Madrid"}
- Categorización: {"id": "S001", "valor": 500, "tipo": "vehículos", "fecha": "2025-03-25", "ubicacion": "Madrid", "categoria": "tráfico"}
- Cálculo: {"id": "S001", "valor": 500, "tipo": "vehículos", "fecha": "2025-03-25", "ubicacion": "Madrid", "promedio_hora": 20.83}
- Corrección: (si "val" fuera "500x") → {"valor": 500}
- Agregación: {"fecha": "2025-03-25", "ubicacion": "Madrid", "vehículos_totales": 1500}

Transformación de la información:

- Comparación: "Madrid tuvo 500 vehículos vs. 600 en Barcelona el 25 de marzo"
- Repercusión: "Un pico de 500 vehículos indica posible congestión"
- Conexión: "El sensor S001 en Madrid registró 500 vehículos"
- Conversación: "El 25 de marzo, Madrid tuvo un tráfico de 500 vehículos según el sensor S001"

Calidad del dato:

- Completitud: Falta "unidad" → Agregar "unidad": "vehículos"
- Credibilidad: Verificar que 500 sea razonable para Madrid.
- Precisión: Asegurar que 500 no sea un redondeo de 498.
- Consistencia: Usar siempre "fecha" en formato "YYYY-MM-DD".
- Interpretabilidad: Cambiar "val" por "vehículos" para claridad.

Impacto en NoSQL: Un JSON bien transformado y de alta calidad reduce el tamaño (evitando redundancia), mejora las consultas y asegura análisis fiables.

6. Relevancia en Big Data y NoSQL

- **Volumen:** La agregación y corrección manejan datos masivos sin colapsar sistemas (ejemplo: límite de 16MB por documento en MongoDB como restricción).
- **Variedad:** La contextualización y categorización unifican datos heterogéneos.
- **Velocidad:** Cálculos predefinidos y consistencia aceleran el procesamiento.
- **Veracidad:** Credibilidad y precisión garantizan confianza en análisis masivos.

Taller C2: Uso Avanzado de JSON para Almacenar Datos

1. ¿Qué es JSON y por qué usarlo para almacenar datos?

JSON (JavaScript Object Notation) es un formato ligero y flexible para representar datos estructurados. Su simplicidad y capacidad para modelar estructuras anidadas lo hacen ideal para aplicaciones modernas que necesitan almacenar información sin un esquema rígido, como en bases de datos NoSQL, APIs o archivos locales.

Ventajas:

- Representa relaciones complejas mediante objetos y arreglos anidados.
- Es legible por humanos y fácilmente procesable por máquinas.
- No depende de una tecnología específica; puede usarse en cualquier sistema.

Ejemplo básico: Almacenar información de un estudiante:

```
{
  "nombre": "Carlos",
  "edad": 22,
  "cursos": [
    {"id": "C01", "nombre": "Matemáticas", "creditos": 4},
    {"id": "C02", "nombre": "Programación", "creditos": 3}
  ]
}
```

2. Almacenamiento de datos con JSON

JSON permite almacenar datos como documentos individuales, ya sea en archivos, APIs o bases de datos. La clave está en estructurar los datos según las necesidades de acceso y uso.

Cómo usarlo:

- **Archivos locales:** Guarda datos en un archivo .json para configuraciones o pequeñas aplicaciones.
- **Intercambio de datos:** Usa JSON como formato en APIs REST para enviar/recibir información.
- **Estructuras dinámicas:** Aprovecha su flexibilidad para agregar o quitar campos sin romper la estructura.

Ejemplo práctico: Un archivo JSON para una biblioteca:

```
{
  "biblioteca": {
    "nombre": "Central",
    "libros": [
      {
        "id": "L001",
        "titulo": "El Quijote",
        "autor": "Cervantes",
        "disponible": true
      },
      {
```

```
        "id": "L002",
        "titulo": "Cien Años de Soledad",
        "autor": "García Márquez",
        "disponible": false
      }
    ]
  }
}
```

3. Modelado de relaciones en JSON

En lugar de tablas relacionales, JSON usa tres patrones principales para modelar relaciones: **embebido**, **referenciado** y **jerárquico**. A continuación, explico cada uno con ejemplos.

a) Patrón Embebido (Embedding)

Los datos relacionados se anidan dentro del mismo objeto. Es útil cuando los datos siempre se usan juntos.

Ejemplo: Un blog con comentarios

```
{
  "post_id": "P001",
  "titulo": "Introducción a JSON",
  "contenido": "Este es el texto del post...",
  "comentarios": [
    {
      "comentario_id": "C001",
      "autor": "Ana",
      "texto": "¡Gran artículo!"
    },
    {
      "comentario_id": "C002",
      "autor": "Luis",
      "texto": "Me aclaró muchas dudas."
    }
  ]
}
```

Ventajas:

- Acceso rápido a todos los datos relacionados.
- Estructura intuitiva para relaciones 1:N pequeñas.

Desventajas:

- Puede crecer demasiado si hay muchos elementos anidados.

b) Patrón Referenciado (Referencing)

Los datos relacionados se almacenan como objetos separados, vinculados por un identificador. Ideal para relaciones complejas o datos que cambian frecuentemente.

Ejemplo: Usuarios y sus publicaciones Usuario:

```
{
  "usuario_id": "U001",
  "nombre": "Ana",
  "publicaciones": ["P001", "P002"]
}
```

Publicación:

```
{
  "post_id": "P001",
  "titulo": "Mi primer post",
  "contenido": "Hola mundo"
}
```

Ventajas:

- Separa los datos, manteniendo los objetos manejables.
- Facilita actualizaciones independientes.

Desventajas:

- Requiere unir los datos manualmente al procesarlos.

c) Patrón Jerárquico (Hierarchical)

Representa estructuras en árbol, como categorías o comentarios anidados. Los datos incluyen referencias o anidamiento para mostrar la jerarquía.

Ejemplo: Árbol de departamentos

```
{
  "departamento_id": "D001",
  "nombre": "Ventas",
  "subdepartamentos": [
    {
      "departamento_id": "D002",
      "nombre": "Ventas Regionales",
      "subdepartamentos": [
        {
          "departamento_id": "D003",
          "nombre": "Ventas Europa"
        }
      ]
    },
    {
      "departamento_id": "D004",
      "nombre": "Ventas Online"
    }
  ]
}
```

Ventajas:

- Natural para estructuras jerárquicas.
- Fácil de navegar en aplicaciones recursivas.

Desventajas:

- Puede complicarse con jerarquías profundas.

4. Problemas de un JSON muy grande

Aunque JSON es flexible, un objeto demasiado grande tiene inconvenientes, especialmente en sistemas con restricciones de tamaño (como MongoDB, que limita los documentos a 16MB).

Problemas:

1. **Límite de tamaño:** Algunos sistemas, como MongoDB, no aceptan documentos mayores a 16MB. Por ejemplo, un JSON con miles de comentarios anidados podría exceder este límite.
    - Ejemplo: Un post con 10,000 comentarios (~1.6KB cada uno) alcanzaría ~16MB.
2. **Rendimiento:** Procesar o transmitir un JSON grande consume más memoria y tiempo.
3. **Mantenimiento:** Actualizar un campo pequeño en un JSON grande requiere reescribir todo el objeto.
4. **Legibilidad:** Un JSON extenso es difícil de inspeccionar manualmente.

Soluciones:

- **División:** Usa el patrón referenciado para separar datos en objetos más pequeños.
- **Paginación:** Almacena solo un subconjunto de datos y usa referencias para el resto (ejemplo: "comentarios": ["C001", "C002", "... ver más"]).
- **Compresión:** Si el JSON se guarda o envía, comprímelo (aunque esto no resuelve límites internos de sistemas).
- **Estructura eficiente:** Minimiza redundancia y usa claves cortas.

Ejemplo de problema:

```
{
  "post_id": "P001",
  "titulo": "Post masivo",
  "comentarios": [ /* 10,000 comentarios */ ]
}
```

Solución: Separar comentarios en otro archivo o estructura:

```
{
  "post_id": "P001",
  "titulo": "Post masivo",
  "comentarios_ref": "comentarios_P001.json"
}
```

5. Ejemplo completo de uso avanzado

Imagina una aplicación de gestión de proyectos con usuarios, tareas y subtareas.

Estructura mixta: Usuario (referenciado):

```
{
  "usuario_id": "U001",
  "nombre": "Ana",
  "proyectos": ["PR001"]
}
```

Proyecto (embebido + jerárquico):

```
{
  "proyecto_id": "PR001",
  "nombre": "Lanzamiento App",
  "tareas": [
    {
      "tarea_id": "T001",
      "descripcion": "Diseñar UI",
      "estado": "completada",
      "subtareas": [
        {"subtarea_id": "ST001", "descripcion": "Crear mockups"}
      ]
    },
    {
      "tarea_id": "T002",
      "descripcion": "Programar backend",
      "estado": "en progreso"
    }
  ]
}
```

Procesamiento:

- Consulta rápida: Todas las tareas de un proyecto están embebidas.
- Escalabilidad: Los usuarios se mantienen separados para evitar documentos gigantes.

6. Conclusión y consejos avanzados

- **Diseño:** Estructura el JSON según cómo accederás a los datos (embebido para acceso frecuente, referenciado para independencia).
- **Tamaño:** Monitorea el crecimiento de los objetos y divide cuando sea necesario.
- **Validación:** Usa herramientas como JSON Schema para asegurar consistencia:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "usuario_id": {"type": "string"},
    "nombre": {"type": "string"}
  },
  "required": ["usuario_id", "nombre"]
}
```

- **Flexibilidad:** Aprovecha la falta de esquema para iterar rápidamente, pero mantén un diseño lógico.