Procesamiento de Grandes Volúmenes de Datos

Conferencia 9: Optimización de Consultas Distribuidas

Deborah Famadas Rodríguez

Universidad de la Habana

9 de noviembre de 2025

El Desafío: Centralizado vs. Distribuido

- **Objetivo Común:** Transformar una consulta de alto nivel (SQL) en una estrategia de ejecución eficiente (Álgebra Relacional).
- **Diferencia Fundamental:** El entorno distribuido introduce una variable de coste dominante: **la comunicación de red**.
- La optimización debe minimizar el coste total, no solo el procesamiento local.

Principio Clave

El coste de la comunicación (ancho de banda, latencia) suele ser órdenes de magnitud superior al coste de CPU y E/S de disco local.

Arquitectura del Procesador de Consultas Distribuidas

1. **Descomposición de la Consulta:** SQL se traduce a un árbol de álgebra relacional.
2. **Localización de Datos:** Determina en qué nodos residen los fragmentos de datos.
3. **Optimización Global:** Fase crucial. Decide qué mover, dónde moverlo y en qué orden.
4. **Optimización Local:** Cada nodo ejecuta su porción del plan de la manera más eficiente.

Ejemplo: `Empleados` (Sitio 1) `JOIN` `Departamento` (Sitio 2)

- `Empleados`: 1,000,000 bytes.
- `Departamento`: 350,000 bytes.
- Resultado (en Sitio 3): 400,000 bytes (asumido).

Estrategia 1: Mover ambos al Sitio 3

Coste: $1{,}000{,}000 + 350{,}000 = \mathbf{1{,}350{,}000}$ **bytes**.

Estrategia 2: Mover `Departamento` al Sitio 1, hacer `JOIN`, enviar resultado

Coste: $350{,}000 + 400{,}000 = \mathbf{750{,}000}$ **bytes**. (Ganadora)

Estrategia 3: Mover `Empleados` al Sitio 2, hacer `JOIN`, enviar resultado

Coste: $1{,}000{,}000 + 400{,}000 = \mathbf{1{,}400{,}000}$ **bytes**.

Ejemplo: `Empleados` (Sitio 1) `JOIN` `Departamento` (Sitio 2)

- `Empleados`: 1,000,000 bytes.
- `Departamento`: 350,000 bytes.
- Resultado (en Sitio 3): 400,000 bytes (asumido).

Estrategia 1: Mover ambos al Sitio 3

Coste: 1,000,000 + 350,000 = **1,350,000 bytes**.

Estrategia 2: Mover `Departamento` al Sitio 1, hacer `JOIN`, enviar resultado

Coste: 350,000 + 400,000 = **750,000 bytes**. (Ganadora)

Estrategia 3: Mover `Empleados` al Sitio 2, hacer `JOIN`, enviar resultado

Coste: 1,000,000 + 400,000 = **1,400,000 bytes**.

Ejemplo: `Empleados` (Sitio 1) `JOIN` `Departamento` (Sitio 2)

- `Empleados`: 1,000,000 bytes.
- `Departamento`: 350,000 bytes.
- Resultado (en Sitio 3): 400,000 bytes (asumido).

Estrategia 1: Mover ambos al Sitio 3

Coste: $1{,}000{,}000 + 350{,}000 =$ **1,350,000 bytes**.

Estrategia 2: Mover `Departamento` al Sitio 1, hacer `JOIN`, enviar resultado

Coste: $350{,}000 + 400{,}000 =$ **750,000 bytes**. (Ganadora)

Estrategia 3: Mover `Empleados` al Sitio 2, hacer `JOIN`, enviar resultado

Coste: $1{,}000{,}000 + 400{,}000 =$ **1,400,000 bytes**.

Ejemplo: `Empleados` (Sitio 1) `JOIN` `Departamento` (Sitio 2)

- `Empleados`: 1,000,000 bytes.
- `Departamento`: 350,000 bytes.
- Resultado (en Sitio 3): 400,000 bytes (asumido).

Estrategia 1: Mover ambos al Sitio 3

Coste: $1{,}000{,}000 + 350{,}000 =$ **1,350,000 bytes**.

Estrategia 2: Mover `Departamento` al Sitio 1, hacer `JOIN`, enviar resultado

Coste: $350{,}000 + 400{,}000 =$ **750,000 bytes**. (Ganadora)

Estrategia 3: Mover `Empleados` al Sitio 2, hacer `JOIN`, enviar resultado

Coste: $1{,}000{,}000 + 400{,}000 =$ **1,400,000 bytes**.

El Objetivo: Minimizar los Datos en Movimiento

- En el ejemplo, la Estrategia 2 ganó (Coste: **750,000 bytes**) porque fue la que *menos datos movió* por la red.
- La Estrategia 3 perdió (Coste: **1,400,000 bytes**) porque movió la tabla grande.
- **El objetivo de esta conferencia:** Explorar las técnicas que usan los motores distribuidos para reducir este coste lo más cerca posible de cero.

El Villano Principal

La operación más costosa que genera este movimiento de datos es el **Data Shuffling**.

El Objetivo: Minimizar los Datos en Movimiento

- En el ejemplo, la Estrategia 2 ganó (Coste: **750,000 bytes**) porque fue la que *menos datos movió* por la red.
- La Estrategia 3 perdió (Coste: **1,400,000 bytes**) porque movió la tabla grande.
- **El objetivo de esta conferencia:** Explorar las técnicas que usan los motores distribuidos para reducir este coste lo más cerca posible de cero.

El Villano Principal

La operación más costosa que genera este movimiento de datos es el **Data Shuffling**.

El Objetivo: Minimizar los Datos en Movimiento

- En el ejemplo, la Estrategia 2 ganó (Coste: **750,000 bytes**) porque fue la que *menos datos movió* por la red.
- La Estrategia 3 perdió (Coste: **1,400,000 bytes**) porque movió la tabla grande.
- **El objetivo de esta conferencia:** Explorar las técnicas que usan los motores distribuidos para reducir este coste lo más cerca posible de cero.

El Villano Principal

La operación más costosa que genera este movimiento de datos es el **Data Shuffling**.

1.2. Anatomía del Data Shuffling

- El **Data Shuffling** es el proceso de redistribuir datos a través de los nodos del clúster.
- Es un prerrequisito fundamental para operaciones como `GROUP BY`, `reduceByKey` y `JOINs`.
- **Objetivo del Shuffle:** Co-localizar todos los registros con la misma clave de unión en el mismo nodo para su procesamiento.
- Es el principal cuello de botella en el procesamiento distribuido.

Fases Mecánicas del Shuffle

1. **Fase de Map (Escritura):**
   - Cada nodo procesa sus datos locales.
   - Aplica una función de particionamiento (ej. hash de la clave).
   - Escribe los datos en búferes y los vierte a archivos temporales en disco local, organizados por destino.
2. **Transferencia de Red:**
   - Fase "todos contra todos" (`all-to-all`).
   - Los nodos *reducers* solicitan los bloques de datos que les corresponden.
3. **Fase de Reduce (Lectura):**
   - Cada *reducer* recibe fragmentos de múltiples *mappers*.
   - Debe fusionar y (a menudo) ordenar estos fragmentos antes de procesar.

El Coste Real del Shuffle

El coste del `shuffle` no es monolítico, es una cascada de cuellos de botella:

- **Sobrecarga de Red:** El coste más visible. Satura el ancho de banda. Puede generar $N \times (N-1)$ conexiones.
- **E/S de Disco:** Los datos intermedios a menudo exceden la RAM y deben "derramarse" (`spilled`) a disco, lo cual es órdenes de magnitud más lento.
- **Coste de CPU y Memoria:**
  - CPU para serializar y deserializar datos.
  - Memoria para búferes de escritura y lectura.
- **Generación de Ficheros Pequeños:** Un `shuffle` con muchas particiones puede sobrecargar el sistema de archivos (ej. HDFS).

Estrategia 1: Broadcast Hash Join (Evitando el Shuffle)

Escenario Ideal

Unir una tabla **grande** con una tabla **significativamente más pequeña**.

1. **Recolección:** El *driver* recolecta la tabla pequeña en su memoria (si es $<$ `spark.sql.autoBroadcastJoinThreshold`).
2. **Difusión (Broadcast):** El *driver* transmite una copia completa de la tabla pequeña a cada *executor*.
3. **Construcción de Tabla Hash:** Cada *executor* recibe la tabla pequeña y construye una tabla hash en su memoria local.
4. **Ejecución Local del Join:** Cada *executor* procesa sus particiones locales de la tabla grande, buscando coincidencias en la tabla hash.

Resultado Clave

¡No hay `shuffle` de la tabla grande!

Estrategia 1: Broadcast Hash Join (Evitando el Shuffle)

Escenario Ideal

Unir una tabla **grande** con una tabla **significativamente más pequeña**.

1. **Recolección:** El *driver* recolecta la tabla pequeña en su memoria (si es $<$ `spark.sql.autoBroadcastJoinThreshold`).
2. **Difusión (Broadcast):** El *driver* transmite una copia completa de la tabla pequeña a cada *executor*.
3. **Construcción de Tabla Hash:** Cada *executor* recibe la tabla pequeña y construye una tabla hash en su memoria local.
4. **Ejecución Local del Join:** Cada *executor* procesa sus particiones locales de la tabla grande, buscando coincidencias en la tabla hash.

Resultado Clave

¡No hay `shuffle` de la tabla grande!

Broadcast Hash Join: Ventajas y Limitaciones

Ventajas:

- **Eliminación del Shuffle:** Ahorra enormes cantidades de E/S de red y disco.
- **Rápido:** Generalmente la estrategia de `JOIN` más rápida.
- Ideal para *star schemas* (tabla de hechos grande + tablas de dimensiones pequeñas).

Limitaciones:

- **Restricción de Memoria:** La tabla pequeña debe caber en la memoria del *driver* Y de cada *executor*.
- **Errores `Out Of Memory` (OOM):** Un riesgo real si las estimaciones de tamaño son incorrectas.
- **Sobrecarga Inicial:** La difusión puede ser un cuello de botella si la tabla "pequeña.^{es} de varios GB.
- No soporta `FULL OUTER JOIN`.

Estrategia 2: Shuffle Sort Merge Join (El Caballo de Batalla)

Escenario

Cuando ambas tablas son grandes y el `Broadcast Join` no es una opción. Es la estrategia por defecto más robusta.

Se ejecuta en tres fases bien definidas:

1. **Fase de Shuffle:** Ambas tablas se reparticionan (barajan) a través del clúster usando un hash de la clave de `JOIN`.
2. **Fase de Sort:** Dentro de cada partición, los datos de cada tabla se ordenan de forma independiente por la clave de `JOIN`.
3. **Fase de Merge:** Con los datos co-localizados y ordenados, el motor usa dos punteros para fusionar los resultados en una sola pasada lineal (similar a *merge sort*).

Shuffle Sort Merge Join: Ventajas y Desventajas

Ventajas:

- **Escalabilidad:** El único método que maneja de forma fiable `JOINs` entre dos tablas de tamaño arbitrariamente grande.
- **Robustez:** Puede "derramar" (`spill`) datos a disco si no caben en memoria, evitando fallos por OOM.

Desventajas:

- **Alto Coste:** Incurre en el coste completo de `shuffle` para **ambas** tablas.
- **Coste de Ordenación:** Requiere un coste computacional adicional para ordenar los datos en cada partición.
- **Vulnerable al Data Skew:** Un desequilibrio en los datos puede ser catastrófico (se verá más adelante).

Tabla Comparativa de Estrategias de Join I

Característica	Broadcast Hash Join	Shuffle Sort Merge Join	Shuffle Hash Join
Movimiento de Datos	Broadcast de tabla pequeña; sin shuffle de tabla grande.	Shuffle completo de ambas tablas.	Shuffle completo de ambas tablas.
Uso de Memoria del Executor	**Alto**. Tabla pequeña completa debe caber en memoria.	**Moderado**. Puede derramar (`spill`) a disco.	**Alto**. Tabla hash de una partición debe caber en memoria.
Requisito de Ordenación	No.	**Sí**. Datos ordenados por clave en cada partición.	No.
Resistencia al Skew	**Alta**. Inmune al skew en la tabla grande.	**Baja**. Muy vulnerable al skew en la clave.	**Baja**. Vulnerable al skew.
Caso de Uso Principal	Unir tabla grande con tabla pequeña (Hechos-Dim).	Unir dos tablas grandes. Estrategia por defecto.	Unir dos tablas de tamaño medio (ordenación costosa).

3.1. ¿Qué es el Data Skew?

- Es una condición en la que los datos se distribuyen de manera desigual entre las particiones del clúster.
- **Key Skew:** Es la forma más perjudicial. Ocurre cuando un pequeño subconjunto de claves (`hot keys`) aparece con una frecuencia desproporcionadamente alta.
- **Ejemplos Comunes:** Claves con valor `NULL`, identificadores de ïnvitado.º "desconocido", o un cliente/producto extremadamente popular.

El Problema

El `shuffle` envía todas las filas con la misma clave al mismo *executor*, sobrecargándolo. La paralelización se rompe.

El Síntoma Inequívoco: Tareas "Straggler"

- Una **tarea** `straggler` **(rezagada)** es aquella que se ejecuta en el *executor* sobrecargado y tarda mucho más en completarse que el resto.
- Una etapa del trabajo no puede finalizar hasta que **todas** sus tareas hayan concluido.
- Una única tarea `straggler` puede detener el progreso de todo el trabajo, desperdiciando recursos del clúster.

Diagnóstico (Ej. Spark UI)

Inspeccionar las métricas de las tareas de una etapa:

Duración "Máxima» ¿ Duración "Mediana.º "Percentil 75"

El Síntoma Inequívoco: Tareas "Straggler"

- Una **tarea** `straggler` **(rezagada)** es aquella que se ejecuta en el *executor* sobrecargado y tarda mucho más en completarse que el resto.
- Una etapa del trabajo no puede finalizar hasta que **todas** sus tareas hayan concluido.
- Una única tarea `straggler` puede detener el progreso de todo el trabajo, desperdiciando recursos del clúster.

Diagnóstico (Ej. Spark UI)

Inspeccionar las métricas de las tareas de una etapa:

Duración "Máxima»¿ Duración "Mediana.º "Percentil 75"

3.2. Mitigación del Skew: Técnica de Salting

Concepto Fundamental

Romper artificialmente la `hot key` en múltiples subclaves para distribuir su carga de trabajo entre varios *executors*. Esto se logra añadiendo un valor aleatorio (el "salt") a la clave.

Proceso de Salting para Joins (Paso a Paso)

1. **Añadir Salt a la Tabla Sesgada (Grande):**
   - Se añade una nueva columna `salt` (ej. `random(0, N-1)`).
   - La nueva clave de `JOIN` es compuesta: `<clave_original, salt>`.
   - `user_id=123` se transforma en `123_0`, `123_1`, ..., `123_(N-1)`.
2. **Replicar la Tabla Pequeña (Búsqueda):**
   - Se ."explota" la tabla pequeña, replicando cada fila N veces.
   - A cada réplica se le añade un valor de `salt` (de 0 a N-1).
   - La fila `user_id=123` ahora tendrá N filas: `123_0`, `123_1`, ...
3. **Ejecutar el Join Salado:**
   - Se realiza el `JOIN` usando la nueva clave compuesta.
   - La carga de la `hot key` se distribuye ahora entre N *executors*.

Consideraciones del Salting

- **Elección del Factor de Salt (N):**
  - Es un compromiso. Un N más alto distribuye mejor la carga, pero también incrementa el tamaño de la tabla pequeña replicada.
  - El objetivo es dividir la partición más grande en fragmentos manejables.
- **Salting para Agregaciones (`GROUP BY`):**
  - Es posible, pero requiere dos pasos:
  - 1. Primera agregación sobre la clave salada (resultados parciales).
  - 2. Segunda agregación sobre la clave original (combinar resultados parciales).
- **Nota Importante:**
  - El `Broadcast Join` es inherentemente resistente al `skew` en la tabla grande, ya que no hace `shuffle` basado en esa clave. Si la tabla pequeña puede ser difundida, ¡es la solución más simple!

Reducción de Datos Pre-Join

Principio

Mover el cómputo a los datos, no los datos al cómputo.

- Estrategias que actúan como un pre-filtro para eliminar datos irrelevantes *antes* de que sean enviados a través de la red.
- Atacan directamente el volumen de datos transferidos durante el `shuffle`.

Técnicas Clave:

- Semijoins
- Bloom Joins

4.1. Semijoins: Filtrando con Datos Reales

- **Definición:** Operación que devuelve las filas de una tabla que tienen al menos una coincidencia en la otra tabla.
- **Problema a Resolver:** Las "tuplas colgantes" (`dangling tuples`) - filas que no encontrarán correspondencia en el `JOIN`.
- Transferir estas tuplas es un desperdicio de red.

Mecanismo de Optimización ($R \bowtie S$ en Sitio 2):

1. **Transferencia de Claves:** Sitio 2 envía *solo* su columna de `JOIN` ($\pi_{key}(S)$) al Sitio 1.
2. **Filtrado Local (Semijoin):** En Sitio 1, se hace un `JOIN` local: $R' = R \ltimes S$.
3. **Transferencia Reducida:** *Solo* se envía R' (reducido) al Sitio 2.
4. **Join Final:** En Sitio 2, se ejecuta el `JOIN` final: $R' \bowtie S$.

4.1. Semijoins: Filtrando con Datos Reales

- **Definición:** Operación que devuelve las filas de una tabla que tienen al menos una coincidencia en la otra tabla.
- **Problema a Resolver:** Las "tuplas colgantes" (`dangling tuples`) - filas que no encontrarán correspondencia en el `JOIN`.
- Transferir estas tuplas es un desperdicio de red.

Mecanismo de Optimización ($R \bowtie S$ en Sitio 2):

1. **Transferencia de Claves:** Sitio 2 envía *solo* su columna de `JOIN` ($\pi_{key}(S)$) al Sitio 1.
2. **Filtrado Local (Semijoin):** En Sitio 1, se hace un `JOIN` local: $R' = R \ltimes S$.
3. **Transferencia Reducida:** *Solo* se envía R' (reducido) al Sitio 2.
4. **Join Final:** En Sitio 2, se ejecuta el `JOIN` final: $R' \bowtie S$.

4.2. Bloom Joins: Filtrando Probabilísticamente

- Una variante del `Semijoin` que usa un **Filtro de Bloom**.
- **Filtro de Bloom:** Estructura de datos probabilística (array de bits) muy compacta.
- Puede determinar si un elemento:
  - "Definitivamente **no** está en un conjunto" (sin falsos negativos).
  - "Posiblemente **sí** está en un conjunto" (permite falsos positivos).

Mecanismo de Optimización ($R \bowtie S$ en Sitio 2):

1. **Creación del Filtro:** Sitio 2 crea un Filtro de Bloom a partir de las claves de S.
2. **Transferencia del Filtro:** Se envía el Filtro de Bloom (muy pequeño) al Sitio 1.
3. **Filtrado Local:** Sitio 1 comprueba cada fila de R contra el filtro.
4. **Transferencia Reducida:** Se envía el R' filtrado (incluye todas las coincidencias reales + algunos falsos positivos) al Sitio 2.
5. **Join Final:** En Sitio 2, se ejecuta $R' \bowtie S$ (los falsos positivos se descartan aquí).

4.2. Bloom Joins: Filtrando Probabilísticamente

- Una variante del `Semijoin` que usa un **Filtro de Bloom**.
- **Filtro de Bloom:** Estructura de datos probabilística (array de bits) muy compacta.
- Puede determinar si un elemento:
  - "Definitivamente **no** está en un conjunto" (sin falsos negativos).
  - "Posiblemente **sí** está en un conjunto" (permite falsos positivos).

Mecanismo de Optimización ($R \bowtie S$ en Sitio 2):

1. **Creación del Filtro:** Sitio 2 crea un Filtro de Bloom a partir de las claves de S.
2. **Transferencia del Filtro:** Se envía el Filtro de Bloom (muy pequeño) al Sitio 1.
3. **Filtrado Local:** Sitio 1 comprueba cada fila de R contra el filtro.
4. **Transferencia Reducida:** Se envía el R' filtrado (incluye todas las coincidencias reales + algunos falsos positivos) al Sitio 2.
5. **Join Final:** En Sitio 2, se ejecuta $R' \bowtie S$ (los falsos positivos se descartan aquí).

Tabla Comparativa: Semijoin vs. Bloom Join I

Característica	Semijoin	Bloom Join
Mecanismo de Filtrado	Determinista (basado en valores de clave reales).	Probabilístico (basado en una estructura de bits hash).
Tamaño de Datos de Filtro	Proporcional a la cardinalidad y tamaño de la clave. Puede ser grande.	Pequeño y constante, configurable. Independiente del tamaño de la clave.
Precisión del Filtrado	100 % preciso. No hay falsos positivos.	Permite **falsos positivos**. No hay falsos negativos.
Coste Computacional	Requiere un `JOIN` local en el sitio de origen.	Requiere creación del filtro y comprobación de hash por fila.
Mejor Escenario de Uso	Claves de `JOIN` pequeñas (ej. enteros) y la precisión es crítica.	Claves de `JOIN` grandes (ej. strings) y se tolera una tasa de falsos positivos.

Modelos de Ejecución

La forma en que un motor maneja los resultados intermedios impacta profundamente el rendimiento.

1. **Modelo de Materialización**
   - Cada operador escribe su resultado completo en disco.
   - El siguiente operador lee ese archivo.
   - **Ejemplo:** Hadoop MapReduce.

2. **Modelo de Pipelining (Iterador)**
   - Los operadores se organizan como iteradores.
   - Las tuplas fluyen de un operador a otro en memoria.
   - **Ejemplo:** Modelo Volcano, estándar moderno.

5.1. Modelo de Materialización

Ventajas:

- **Robustez y Escalabilidad:** No está limitado por la RAM. Maneja resultados intermedios masivos.
- **Recuperación de Fallos:** Los archivos en disco actúan como puntos de control (*checkpoints*).

Desventajas:

- **Alto Coste de E/S de Disco:** Escritura y lectura repetida desde disco.
- **Alta Latencia:** Extremadamente lento comparado con el acceso a memoria.

5.2. Modelo de Pipelining (Iterador)

Ventajas:

- **Rendimiento Mejorado:** Elimina la costosa E/S de disco.
- **Baja Latencia:** Puede empezar a producir resultados ("Time to First Row") mucho antes.

Desventajas:

- **Presión sobre la Memoria:** Requiere búferes de memoria.
- **Pipeline Breakers:** Operadores como `SORT` o `HASH JOIN` (fase de construcción) deben consumir toda la entrada antes de producir salida, forzando una materialización (idealmente en memoria).

5.3. Optimización Extrema: Whole-Stage Code Generation

- El modelo iterador "tupla a tupla"sufre de sobrecarga por llamadas a funciones virtuales (`next()`) y mal uso de la caché de CPU.
- **Solución:** `Whole-Stage Code Generation` (WSCG)
- **Ejemplo:** Motor Tungsten de Apache Spark.

¿Cómo funciona?

- En lugar de interpretar cada operador, WSCG examina una secuencia de operadores (una .^{et}apa completa").
- Genera dinámicamente código de bytes de Java optimizado para esa secuencia específica.
- Fusiona múltiples operaciones (ej. filtro + proyección) en un **único bucle `for` de Java**.
- Opera sobre datos binarios `off-heap` (fuera del heap de JVM), eliminando el *Garbage Collection* y la sobrecarga de objetos.

Resultado

Maximiza la eficiencia de la CPU, acercando el rendimiento al de un

5.3. Optimización Extrema: Whole-Stage Code Generation

- El modelo iterador "tupla a tupla"sufre de sobrecarga por llamadas a funciones virtuales (`next()`) y mal uso de la caché de CPU.
- **Solución:** `Whole-Stage Code Generation` (WSCG)
- **Ejemplo:** Motor Tungsten de Apache Spark.

¿Cómo funciona?

- En lugar de interpretar cada operador, WSCG examina una secuencia de operadores (una .^{et}apa completa").
- Genera dinámicamente código de bytes de Java optimizado para esa secuencia específica.
- Fusiona múltiples operaciones (ej. filtro + proyección) en un **único bucle `for` de Java**.
- Opera sobre datos binarios `off-heap` (fuera del heap de JVM), eliminando el *Garbage Collection* y la sobrecarga de objetos.

Resultado

Maximiza la eficiencia de la CPU, acercando el rendimiento al de un

Diseño Físico Proactivo

- Las estrategias de optimización más efectivas se aplican *antes* de que la consulta se ejecute: en el diseño físico del almacenamiento.
- Decisiones sobre cómo se organizan, agrupan y localizan los datos.
- Pueden **eliminar por completo** las operaciones más costosas, como el `shuffle`.

6.1. Particionamiento y Bucketing

- **Particionamiento (de Directorios):**
  - Divide la tabla en subdirectorios basados en columnas de baja cardinalidad (ej. `fecha`, `region`).
  - Permite "poda de particiones" (`partition pruning`): el motor ignora los directorios que no coinciden con el filtro `WHERE`.
  - **Optimiza: Filtros.**
- **Bucketing (o Clustering):**
  - Divide los datos en un número fijo de archivos (`buckets`) basado en el **hash** de una columna (ej. `user_id`).
  - **Optimiza: JOINs.**

El "Superpoder" del Bucketing

Si dos tablas grandes están *bucketeadas* por la misma clave (`user_id`) y tienen el mismo número de `buckets`, ¡el `shuffle` se evita! El motor sabe que el `bucket` 5 de la tabla A solo puede coincidir con el `bucket` 5 de la tabla B. Es un `shuffle` pre-calculado.

6.1. Particionamiento y Bucketing

- **Particionamiento (de Directorios):**
  - Divide la tabla en subdirectorios basados en columnas de baja cardinalidad (ej. `fecha`, `region`).
  - Permite "poda de particiones" (`partition pruning`): el motor ignora los directorios que no coinciden con el filtro `WHERE`.
  - **Optimiza: Filtros.**
- **Bucketing (o Clustering):**
  - Divide los datos en un número fijo de archivos (`buckets`) basado en el **hash** de una columna (ej. `user_id`).
  - **Optimiza: JOINs.**

El "Superpoder" del Bucketing

Si dos tablas grandes están *bucketeadas* por la misma clave (`user_id`) y tienen el mismo número de `buckets`, ¡el `shuffle` se evita! El motor sabe que el `bucket` 5 de la tabla A solo puede coincidir con el `bucket` 5 de la tabla B. Es un `shuffle` pre-calculado.

6.2. Índices Globales

- **Problema:** ¿Cómo encontrar un `transaction_id` específico en una tabla particionada por fecha sin escanear todo?
- **Índice Local:** Un índice dentro de cada partición. Aún requiere consultar cada partición.
- **Índice Global:** Una estructura de datos única que abarca *todas* las particiones.
- Mapea un valor de clave de índice (ej. `transaction_id`) a su ubicación física exacta (nodo y partición).

Ventaja (Lectura):

- Búsquedas puntuales (`point lookups`) de baja latencia.
- Evita `full table scans`.

Coste (Escritura):

- Ralentiza `INSERT`, `UPDATE`, `DELETE`.
- Tanto los datos como el índice global deben actualizarse de forma consistente.

6.2. Índices Globales

- **Problema:** ¿Cómo encontrar un `transaction_id` específico en una tabla particionada por fecha sin escanear todo?
- **Índice Local:** Un índice dentro de cada partición. Aún requiere consultar cada partición.
- **Índice Global:** Una estructura de datos única que abarca *todas* las particiones.
- Mapea un valor de clave de índice (ej. `transaction_id`) a su ubicación física exacta (nodo y partición).

Ventaja (Lectura):

- Búsquedas puntuales (`point lookups`) de baja latencia.
- Evita `full table scans`.

Coste (Escritura):

- Ralentiza `INSERT`, `UPDATE`, `DELETE`.
- Tanto los datos como el índice global deben actualizarse de forma consistente.

6.3. Z-Ordering

- Una técnica de optimización de diseño que co-localiza datos relacionados en los mismos archivos, pero de forma **multidimensional**.
- Combina los bits de múltiples columnas (ej. `latitud` y `longitud`) para crear un valor único (siguiendo una curva Z o `Z-order curve`).
- Los datos se ordenan y agrupan basándose en este valor compuesto.

Beneficio

Cuando una consulta filtra por un rango en *varias* de las columnas del Z-Order, el motor puede "saltarse" (`skip`) un número mucho mayor de archivos, ya que los datos relevantes estarán concentrados en un conjunto más pequeño y contiguo de archivos.

6.3. Z-Ordering

- Una técnica de optimización de diseño que co-localiza datos relacionados en los mismos archivos, pero de forma **multidimensional**.
- Combina los bits de múltiples columnas (ej. `latitud` y `longitud`) para crear un valor único (siguiendo una curva Z o `Z-order curve`).
- Los datos se ordenan y agrupan basándose en este valor compuesto.

Beneficio

Cuando una consulta filtra por un rango en *varias* de las columnas del Z-Order, el motor puede "saltarse" (`skip`) un número mucho mayor de archivos, ya que los datos relevantes estarán concentrados en un conjunto más pequeño y contiguo de archivos.

De la Optimización Estática a la Reactiva

- **Optimización Estática (CBO - Cost-Based Optimizer):**
  - Toma todas las decisiones *antes* de la ejecución.
  - Se basa en **estadísticas** pre-calculadas (tamaño, cardinalidad, etc.).
  - **Problema:** Las estadísticas pueden ser inexactas, estar desactualizadas o no existir para resultados intermedios.
  - Consecuencia: Generación de planes subóptimos.
- **Optimización Reactiva (AQE - Adaptive Query Execution):**
  - Capacidad del motor para **re-optimizar planes** *durante* el tiempo de ejecución.
  - Utiliza estadísticas precisas y en tiempo real de los resultados intermedios.

De la Optimización Estática a la Reactiva

- **Optimización Estática (CBO - Cost-Based Optimizer):**
  - Toma todas las decisiones *antes* de la ejecución.
  - Se basa en **estadísticas** pre-calculadas (tamaño, cardinalidad, etc.).
  - **Problema:** Las estadísticas pueden ser inexactas, estar desactualizadas o no existir para resultados intermedios.
  - Consecuencia: Generación de planes subóptimos.
- **Optimización Reactiva (AQE - Adaptive Query Execution):**
  - Capacidad del motor para **re-optimizar planes** *durante* el tiempo de ejecución.
  - Utiliza estadísticas precisas y en tiempo real de los resultados intermedios.

El Paradigma de AQE

Modelo Tradicional: "Planificar y Ejecutar"

Plan $\rightarrow$ Ejecutar

Modelo AQE: .Ejecutar, Medir, Re-planificar"

Ejecutar Etapa 1 $\rightarrow$ Medir Salida $\rightarrow$ Re-planificar Etapa 2 $\rightarrow$ Continuar...

- El motor ejecuta la consulta por etapas (divididas por `shuffle` o `broadcast`).
- En estos puntos, tiene estadísticas 100 % precisas sobre el tamaño real de los datos.
- El plan de consulta .evoluciona.a medida que se ejecuta.

El Paradigma de AQE

Modelo Tradicional: "Planificar y Ejecutar"

Plan $\rightarrow$ Ejecutar

Modelo AQE: ."Ej"ecutar, Medir, Re-planificar"

Ejecutar Etapa 1 $\rightarrow$ Medir Salida $\rightarrow$ Re-planificar Etapa 2 $\rightarrow$ Continuar...

- El motor ejecuta la consulta por etapas (divididas por `shuffle` o `broadcast`).
- En estos puntos, tiene estadísticas 100 % precisas sobre el tamaño real de los datos.
- El plan de consulta ."ev"oluciona."a" medida que se ejecuta.

7.3. Capacidades Clave de AQE (en Spark)

- **Cambio Dinámico de Estrategias de Join:**
  - El CBO planifica un `Shuffle Sort Merge Join`.
  - Una etapa de filtrado reduce drásticamente una tabla.
  - AQE lo detecta y cambia dinámicamente a un `Broadcast Hash Join` mucho más rápido.
- **Coalescencia Dinámica de Particiones:**
  - Un `shuffle` produce muchas particiones muy pequeñas (ineficiente).
  - AQE las detecta y las **fusiona automáticamente** en un número menor de particiones de tamaño óptimo.
- **Manejo Dinámico de Skew Join:**
  - AQE detecta `skew` a partir de las estadísticas del `shuffle`.
  - **Divide automáticamente** la partición grande en sub-particiones.
  - Replica las filas correspondientes de la otra tabla para unirse a estas sub-particiones.
  - ¡Mitiga el `skew` sin `salting` manual!

7.3. Capacidades Clave de AQE (en Spark)

- **Cambio Dinámico de Estrategias de Join:**
  - El CBO planifica un `Shuffle Sort Merge Join`.
  - Una etapa de filtrado reduce drásticamente una tabla.
  - AQE lo detecta y cambia dinámicamente a un `Broadcast Hash Join` mucho más rápido.
- **Coalescencia Dinámica de Particiones:**
  - Un `shuffle` produce muchas particiones muy pequeñas (ineficiente).
  - AQE las detecta y las **fusiona automáticamente** en un número menor de particiones de tamaño óptimo.
- **Manejo Dinámico de Skew Join:**
  - AQE detecta `skew` a partir de las estadísticas del `shuffle`.
  - **Divide automáticamente** la partición grande en sub-particiones.
  - Replica las filas correspondientes de la otra tabla para unirse a estas sub-particiones.
  - ¡Mitiga el `skew` sin `salting` manual!

7.3. Capacidades Clave de AQE (en Spark)

- **Cambio Dinámico de Estrategias de Join:**
  - El CBO planifica un `Shuffle Sort Merge Join`.
  - Una etapa de filtrado reduce drásticamente una tabla.
  - AQE lo detecta y cambia dinámicamente a un `Broadcast Hash Join` mucho más rápido.
- **Coalescencia Dinámica de Particiones:**
  - Un `shuffle` produce muchas particiones muy pequeñas (ineficiente).
  - AQE las detecta y las **fusiona automáticamente** en un número menor de particiones de tamaño óptimo.
- **Manejo Dinámico de Skew Join:**
  - AQE detecta `skew` a partir de las estadísticas del `shuffle`.
  - **Divide automáticamente** la partición grande en sub-particiones.
  - Replica las filas correspondientes de la otra tabla para unirse a estas sub-particiones.
  - ¡Mitiga el `skew` sin `salting` manual!

¿Preguntas?