Índices y Estrategias de Optimización de Consultas - Día 2

Change Status

pending · 40 min

Learning Objectives

1. Seleccionar índices apropiados para diferentes patrones de consulta analítica
2. Interpretar planes de ejecución para identificar cuellos de botella
3. Implementar estrategias de particionamiento efectivas para datasets grandes

Activities and Learning

Task 1: Tipos de Índices y su Aplicación Estratégica (10 minutos)

Los índices son estructuras de datos que aceleran consultas, pero elegir el índice correcto requiere entender los patrones de consulta específicos del análisis.

Índices B-Tree: El Trabajo Pesado

Estructura: Árbol balanceado que mantiene datos ordenados para búsquedas eficientes.

Casos de uso ideales:

Búsquedas por rango: `WHERE fecha BETWEEN '2024-01-01' AND '2024-12-31'`
Ordenamiento: `ORDER BY fecha_venta DESC`
Joins: Columnas usadas en `ON` clauses
Filtros selectivos: `WHERE status = 'completed'`

```
-- Índices B-Tree para consultas analíticas comunes
CREATE INDEX idx_ventas_fecha ON hechos_ventas(fecha_venta);
CREATE INDEX idx_ventas_cliente_fecha ON hechos_ventas(id_cliente, fecha_venta);
CREATE INDEX idx_ventas_producto_categoria ON dim_producto(id_categoria, precio);

-- Ejemplo de uso efectivo
EXPLAIN ANALYZE
SELECT SUM(total_venta), AVG(total_venta)
FROM hechos_ventas
WHERE fecha_venta BETWEEN '2024-01-01' AND '2024-12-31'
  AND id_cliente = 12345;
```

Índices Bitmap: Eficiencia en Datos Categóricos

Estructura: Vectores de bits para cada valor único, ideales para columnas con baja cardinalidad.

Casos de uso ideales:

Columnas categóricas: Status, categorías, tipos
Múltiples condiciones AND/OR: `WHERE region IN ('Madrid', 'Barcelona') AND segmento = 'VIP'`
Data warehousing: Filtros complejos en dimensiones

```
-- Índices bitmap para dimensiones categóricas
CREATE BITMAP INDEX idx_cliente_segmento ON dim_cliente(segmento);
CREATE BITMAP INDEX idx_cliente_region ON dim_cliente(region);
CREATE BITMAP INDEX idx_venta_status ON hechos_ventas(status_venta);

-- Ejemplo: Consulta con múltiples filtros categóricos
EXPLAIN ANALYZE
SELECT COUNT(*), SUM(total_venta)
FROM hechos_ventas hv
JOIN dim_cliente dc ON hv.id_cliente = dc.id
WHERE dc.segmento = 'VIP'
  AND dc.region IN ('Madrid', 'Barcelona', 'Valencia')
  AND hv.status_venta = 'completed';
```

Índices Compuestos: Optimización de Consultas Multicolumna

Estructura: Índice en múltiples columnas, efectivo cuando se consultan juntas.

Orden importa: La primera columna debe ser la más selectiva o usada sola.

```
-- Índice compuesto para consultas analíticas típicas
CREATE INDEX idx_ventas_analisis ON hechos_ventas(id_tiempo, id_cliente, id_producto);

-- Bueno para consultas que filtran por tiempo, luego cliente, luego producto
EXPLAIN ANALYZE
```

```
SELECT SUM(total_venta), COUNT(*)
FROM hechos_ventas
WHERE id_tiempo BETWEEN 20240101 AND 20241231  -- Rango de tiempo
  AND id_cliente = 12345                       -- Cliente específico
  AND id_producto IN (100, 200, 300);          -- Productos específicos

-- Mal orden para consultas que empiezan por cliente
-- CREATE INDEX idx_mal_orden ON hechos_ventas(id_cliente, id_tiempo, id_producto);
```

Índices Especializados: Funcionales y Parciales

Índices funcionales: Aceleran expresiones calculadas.

```
-- Índice en expresión calculada
CREATE INDEX idx_ventas_mes ON hechos_ventas(EXTRACT(YEAR FROM fecha_venta), EXTRACT(MONTH FROM fecha_venta));

-- Útil para consultas agrupadas por mes
SELECT EXTRACT(YEAR FROM fecha_venta) as año,
       EXTRACT(MONTH FROM fecha_venta) as mes,
       SUM(total_venta)
FROM hechos_ventas
GROUP BY EXTRACT(YEAR FROM fecha_venta), EXTRACT(MONTH FROM fecha_venta);
```

Índices parciales: Solo indexan filas que cumplen condición.

```
-- Índice solo para ventas grandes (útil para análisis de outliers)
CREATE INDEX idx_ventas_grandes ON hechos_ventas(total_venta)
WHERE total_venta > 1000;

-- Índice solo para clientes activos
CREATE INDEX idx_clientes_activos ON dim_cliente(email)
WHERE activo = true AND ultima_compra > '2023-01-01';
```

Task 2: Estrategias de Particionamiento para Datasets Grandes (10 minutos)

El particionamiento divide tablas grandes en piezas manejables, mejorando performance y mantenibilidad.

Particionamiento por Rangos (Range Partitioning)

Estrategia: Divide datos por rangos de valores continuos.

Ideal para: Datos temporales, numéricos con rangos naturales.

```
-- Particionamiento por rangos para datos temporales
CREATE TABLE hechos_ventas (
    id_venta INTEGER,
    fecha_venta DATE,
    -- ... otras columnas
) PARTITION BY RANGE (fecha_venta);

-- Crear particiones mensuales
CREATE TABLE hechos_ventas_y2024m01 PARTITION OF hechos_ventas
    FOR VALUES FROM ('2024-01-01') TO ('2024-02-01');

CREATE TABLE hechos_ventas_y2024m02 PARTITION OF hechos_ventas
    FOR VALUES FROM ('2024-02-01') TO ('2024-03-01');

-- Ventajas: Eliminación automática de particiones antiguas
ALTER TABLE hechos_ventas DETACH PARTITION hechos_ventas_y2023m12;
```

Particionamiento por Lista (List Partitioning)

Estrategia: Divide datos por valores específicos de categorías.

Ideal para: Columnas categóricas con grupos lógicos.

```
-- Particionamiento por lista para regiones geográficas
CREATE TABLE hechos_ventas PARTITION BY LIST (region);

CREATE TABLE ventas_espana PARTITION OF hechos_ventas
    FOR VALUES IN ('Madrid', 'Barcelona', 'Valencia', 'Sevilla');

CREATE TABLE ventas_latam PARTITION OF hechos_ventas
    FOR VALUES IN ('Mexico', 'Argentina', 'Colombia', 'Chile');

CREATE TABLE ventas_eeuu PARTITION OF hechos_ventas
    FOR VALUES IN ('California', 'Texas', 'Florida', 'Nueva York');
```

Particionamiento por Hash (Hash Partitioning)

Estrategia: Distribuye datos uniformemente usando función hash.

Ideal para: Distribución uniforme, escalabilidad horizontal.

```
-- Particionamiento hash para distribución uniforme
CREATE TABLE hechos_ventas PARTITION BY HASH (id_cliente);

-- Crear particiones hash (PostgreSQL moderno)
CREATE TABLE ventas_hash_01 PARTITION OF hechos_ventas
    FOR VALUES WITH (MODULUS 4, REMAINDER 0);

CREATE TABLE ventas_hash_02 PARTITION OF hechos_ventas
    FOR VALUES WITH (MODULUS 4, REMAINDER 1);

CREATE TABLE ventas_hash_03 PARTITION OF hechos_ventas
    FOR VALUES WITH (MODULUS 4, REMAINDER 2);

CREATE TABLE ventas_hash_04 PARTITION OF hechos_ventas
    FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

Estrategias de Particionamiento Compuesto

Subparticionamiento: Combinar estrategias para granularidad fina.

```
-- Particionamiento compuesto: Range por tiempo, luego Hash por cliente
CREATE TABLE hechos_ventas PARTITION BY RANGE (fecha_venta);

CREATE TABLE ventas_2024 PARTITION OF hechos_ventas
    FOR VALUES FROM ('2024-01-01') TO ('2025-01-01')
    PARTITION BY HASH (id_cliente);

-- Subparticiones
CREATE TABLE ventas_2024_q1 PARTITION OF ventas_2024
    FOR VALUES FROM ('2024-01-01') TO ('2024-04-01')
    PARTITION BY HASH (id_cliente);
```

Task 3: Análisis de Planes de Ejecución y Optimización (10 minutos)

Los planes de ejecución revelan cómo el optimizador procesa consultas, permitiendo identificar cuellos de botella y oportunidades de optimización.

Interpretación de EXPLAIN ANALYZE

Componentes clave del plan:

Cost: Estimación de recursos (CPU + I/O)
Rows: Número estimado de filas procesadas
Width: Ancho promedio de filas en bytes
Actual Time: Tiempo real de ejecución

```
-- Ejemplo de análisis de plan de ejecución
EXPLAIN ANALYZE
SELECT
    dt.año, dt.mes,
    SUM(hv.total_venta) as ventas_total,
    COUNT(DISTINCT hv.id_cliente) as clientes_unicos
FROM hechos_ventas hv
JOIN dim_tiempo dt ON hv.id_tiempo = dt.id
WHERE dt.año = 2024
GROUP BY dt.año, dt.mes
ORDER BY dt.año, dt.mes;

-- Plan esperado (con índices apropiados):
-- HashAggregate (cost=1000.00..1000.50 rows=12 width=32)
--   Group Key: dt.año, dt.mes
--   -> Sort (cost=999.00..999.50 rows=200 width=16)
--     Sort Key: dt.año, dt.mes
--     -> Hash Join (cost=100.00..900.00 rows=10000 width=16)
--       -> Seq Scan on hechos_ventas hv (cost=0.00..800.00 rows=10000 width=12)
--       -> Hash (cost=50.00..50.00 rows=365 width=8)
--         -> Index Scan on dim_tiempo (cost=0.00..50.00 rows=365 width=8)
```

Problemas Comunes y Soluciones

Sequential Scan cuando debería usar índice:

```
-- Problema: Seq Scan on large_table (cost=10000.00..20000.00 rows=1000000 width=100)
-- Solución: Crear índice apropiado
CREATE INDEX idx_table_columna ON large_table(columna_filtrada);
```

Nested Loop ineficiente:

```
-- Problema: Nested Loop (cost=100000.00..1000000.00 rows=10000 width=200)
-- Solución: Estadísticas actualizadas o índices en columnas de join
ANALYZE large_table;
CREATE INDEX idx_join_column ON large_table(join_column);
```

Hash Join vs Merge Join vs Nested Loop:

Hash Join: Bueno para joins grandes sin índices ordenados

Merge Join: Ideal cuando ambas tablas están ordenadas
Nested Loop: Eficiente solo con datasets pequeños