Dicas de Projeto Físico Relacional

- Importância desta etapa
  - definição de estratégias de acesso para maximizar o desempenho

- Estratégias de acesso
  - devem ser constantemente revistas e ajustadas pelo DBA durante o tempo de vida do BD (*Tuning* do BD)
    - análise de processamento de operações sobre o BD e da organização/volume dos dados

Implementação de Consultas

- Definir expressões SQL com melhor desempenho possível
  - processadores de consultas de SGBDs podem ter capacidades limitadas de otimização
- Exemplos

```
SELECT f.título, a.nomeArt
FROM Filmes f, Elenco e, Atores a
WHERE e.filme = f.id e e.ator = a.código
```

otimização

```
SELECT f.título, a.nomeArt
FROM Filmes f JOIN Elenco e ON e.filme = f.id
     JOIN Atores a ON e.ator = a.código
```

Implementação de Consultas

- Definir condições SQL com melhor desempenho possível
  - filtragens AND são melhores que filtragens OR
  - filtragens NOT(predicado) devem ser evitadas

a) $\neg\ (p1 \vee p2) \equiv (\neg\ p1) \wedge (\neg\ p2)$

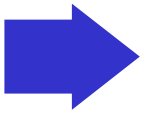
b) $\neg\ (p1 \wedge p2) \equiv (\neg\ p1) \vee (\neg\ p2)$

c) $p1 \vee (p2 \wedge p3) \equiv (p1 \vee p2) \wedge (p1 \vee p3)$

d) $p1 \wedge (p2 \vee p3) \equiv (p1 \wedge p2) \vee (p1 \wedge p3)$

  - exemplo (item a)

```
SELECT *
FROM Fitas ft JOIN Filmes f
ON ft.filme = f.numero
WHERE NOT
   (f.numero = 10 OR
   ft.locadoPara IS NULL)
```

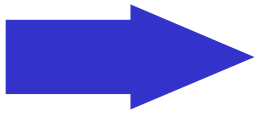


```
SELECT *
FROM Fitas ft JOIN Filmes f
ON ft.filme = f.numero
WHERE f.numero <> 10
AND
ft.locadoPara IS NOT NULL
```

Implementação de Consultas

- Definir predicados SQL com melhor desempenho possível
  - ordem dos predicados de uma condição muitas vezes é relevante
    - exemplo: utilize **`p1`** `OR p2` ao invés de `p2 OR p1` se o predicado **`p1`** é mais provável de ser verdadeiro
  - Valores constantes geram filtros mais seletivos
    - exemplo:

```
SELECT *
 FROM ...
 WHERE a.x = b.y
 ...
 AND b.y = 100
```



```
SELECT *
FROM ...
 WHERE a.x = 100
...
```

Índices

- Vantagens do uso de índices
  - mantém apontadores físicos
    - maior rapidez no acesso a dados
  - são estruturas de dados mais compactas
    - grande parte fica na memória principal
    - agilizam o processamento de predicados de igualdade por atributos indexados
      - teste feito em memória
- Quando se recomenda o uso de um índice
  - atributos envolvidos em predicados de seleção ou de junção em consultas frequentes
  - atributos de tabelas com grande volume estimado de dados e passíveis de consultas com certa frequência

Índices

- Quando se recomenda o uso de índices
  - atributos que são chaves alternativas
    - estrutura do índice é menor
      - » evita listas de apontadores auxiliares para acessar cada ocorrência de valor, no caso de atributo não-chave
  - atributos envolvidos em transações "críticas"
    - exigem tempo de resposta muito pequeno
- Quando não se recomenda o uso de índices
  - atributos que sofrem muita atualização e pouca consulta
    - exigem atualização frequente da estrutura de índices
- Índices compostos
  - geram estruturas mais complexas de gerenciar
  - recomendados apenas quando dois ou mais atributos são testados sempre em conjunto
    - exemplo: consulta ao estoque de calçados de uma loja
      - verificação conjunta de *(tipo_calçado, numeração)*

Índices

- Índices *B-Tree*
  - indicados para atributos não-chave e para predicados de desigualdade (<, >=, intervalo de valores, etc)
    - algoritmos baseados em buscas sobre sub-árvores que mantêm dados ordenados
- Índices *hash*
  - indicados para predicados de igualdade ou junções envolvendo atributos chave ou `UNIQUE`
    - acesso a poucos blocos de índice
  - indicados para tabelas dinâmicas
    - crescem e encolhem com frequência
    - reorganização de estruturas *hash* é rápida