



PostgreSQL Tuning

Rodrigo Hjort
SERPRO/CETEC



O Serviço Federal de Processamento de Dados - SERPRO é uma empresa pública vinculada ao Ministério da Fazenda.

Foi criada em 1964 com o objetivo de modernizar e dar agilidade a setores estratégicos da Administração Pública brasileira prestando serviços em Tecnologia da Informação e Comunicações.



1. O que afeta a performance de um servidor?
2. Ajustando parâmetros no servidor
3. Introdução aos índices
4. Utilizando índices no PostgreSQL
5. Otimizando as consultas SQL
6. Analisando os planos de acesso
7. Buscas textuais (Full Text Search)
8. Técnicas para carga em massa

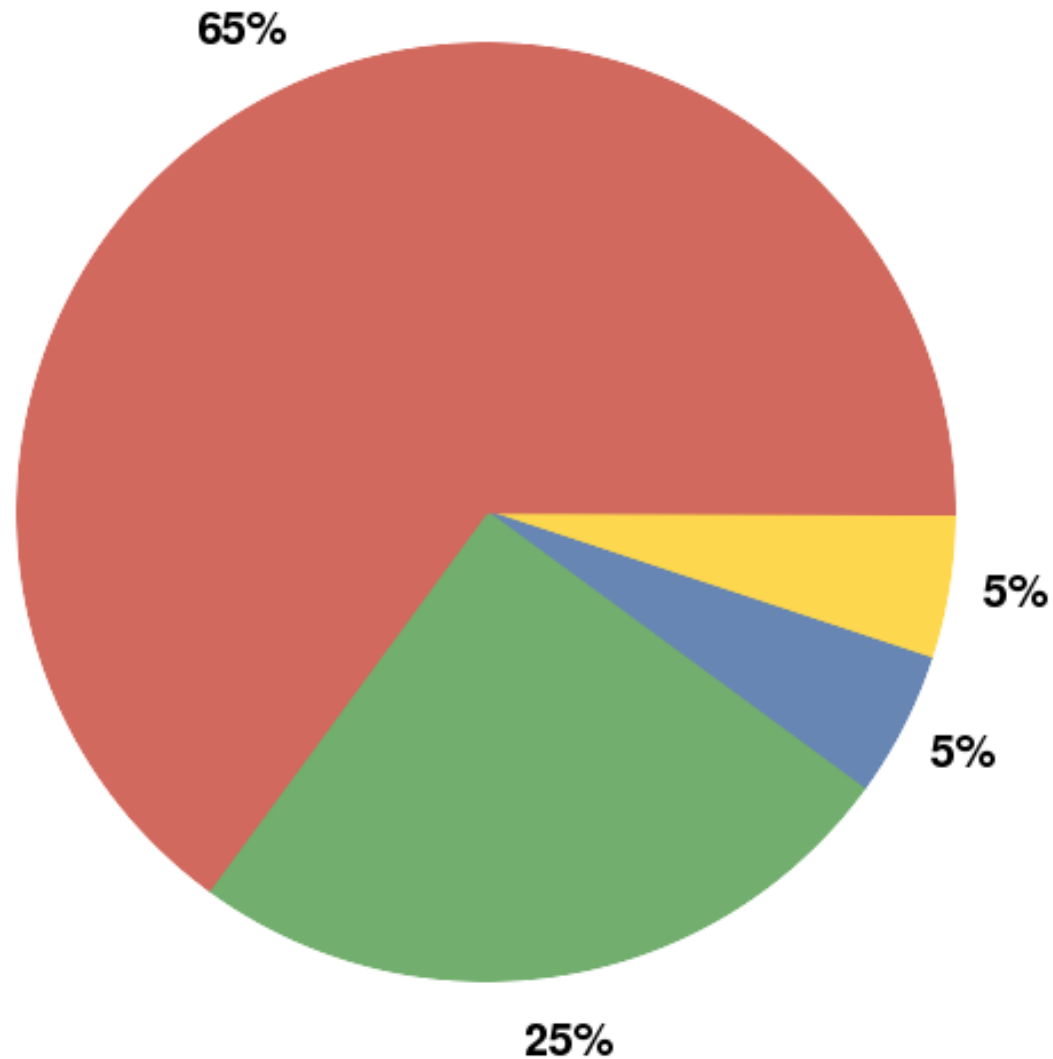


O que afeta a performance de um servidor?



Causas de baixo desempenho

■ SO / Rede ■ SGBD ■ Modelagem ■ Aplicação

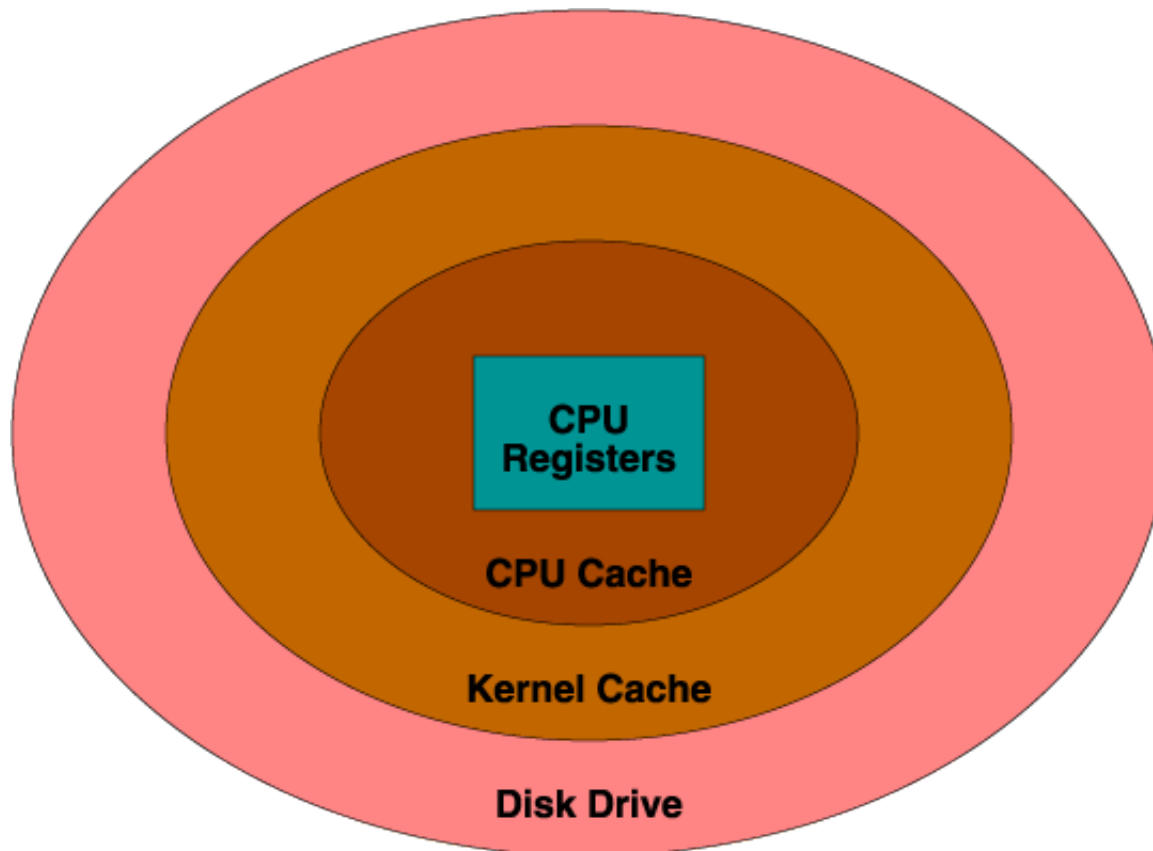


Fonte: SQL Magazine



Níveis de cache

- quanto mais próxima da CPU, menor a capacidade de armazenamento e maior a velocidade de acesso à informação



Dispositivo	Unid.
Registrador	Bytes
Cache CPU	kB
Memória RAM	MB
Disco rígido	GB



Como funciona o cache?

- o objetivo do mecanismo de cache é manter em memória os dados mais requisitados para evitar acesso físico
- acesso ao disco é 100 mil vezes mais lento que em memória!
- **cache hit**: ocorre quando a informação requerida é encontrada na memória, poupando acesso ao disco rígido
- **cache miss**: ocorre quando a informação já não está mais na memória e por isso uma leitura física é provocada



- um sistema gerenciador é um grande consumidor de recursos como:
 - processamento (CPU)
 - memória (RAM)
 - disco rígido (HD)
- o grande desafio para obter um bom desempenho em SGBDs é
 - utilizar da melhor forma possível os recursos de hardware, mantendo em memória as informações mais utilizadas a fim de se evitar acesso físico ao disco



Ajustando parâmetros no servidor



- antes de qualquer modificação nos parâmetros do SGBD:
 - kernel do SO mais adequado para o hardware:
 - arquitetura (x86 – 32 bits, x86_64 – 64 bits)
 - processamento múltiplo (SMP)
 - sistema de arquivos mais adequado
 - escolher entre XFS, ReiserFS, ext3, ...
 - assegurar que não existem problemas de hardware ou software ocorrendo no servidor



Alterando os parâmetros



- arquivo de configurações do PostgreSQL:
`/etc/postgresql/8.3/main/postgresql.conf`
- alguns parâmetros exigem reinicialização:
`/etc/init.d/postgresql-8.3 restart`
- para a maioria basta enviar um sinal:
`/etc/init.d/postgresql-8.3 reload`



- *max_connections = <num>*
- número máximo de conexões ao servidor
- manter somente o necessário!
- espaços para os backends são alocados mesmo quando conexões não são usadas
- Default: 100
- Sugestão: calcular para cada caso



- *shared_buffers = <num> MB*
- parâmetro mas importante no tuning!
- indica quantas páginas (de 8kB) serão utilizadas da memória real da máquina
- o default é extremamente baixo: 24 MB!
- existe um valor ótimo que poderá causar efeitos contrários se for ultrapassado



Quanto usar de memória?



- não é 100%: o SO e os outros processos também precisam de memória
- é preciso deixar área livre para operações de ordenação (sort) e manutenção (work)
- o PostgreSQL faz uso do cache do sistema de arquivos e do sistema operacional
- se o servidor for dedicado (≥ 1 GB RAM): alocar de 25% a 40% da memória total disponível



Como calcular o valor?

- verifique o total de memória disponível:
`$ free -b`
- calcule a quantidade a reservar:
`$ echo "986976256 * .4" | bc`
- calcule o número de blocos necessários:
`$ echo "394790502 / 8192" | bc`
- altere os seguintes parâmetros:
 - kernel.shmmax: na configuração do Linux
 - shared_buffers: na configuração do PostgreSQL



Como saber se é suficiente?



- *cache hit ratio*: indica o percentual de uso efetivo da memória pelo PostgreSQL

- para calcular essa taxa, execute:

```
SELECT trunc(sum(blks_hit) /  
             sum(blks_read + blks_hit) * 100, 2)  
AS cache_hit_ratio  
FROM pg_stat_database;
```

- é preciso habilitar as estatísticas!



- *work_mem* = *<num> MB*
- memória para operações internas:
 - ordenação: ORDER BY, DISTINCT, merge joins
 - tabelas de hash: IN (...), hash joins, agregações
- uma única consulta pode iniciar diversas operações em paralelo que utilizam cada uma um número múltiplo desse valor
- ao passar desse valor serão usados arquivos temporários em disco (swap)!
- Default: 1MB, Sugestão: 32MB



Um espaço para manutenções

- *maintenance_work_mem* = *<num> MB*
- memória para operações de manutenção:
 - **VACUUM, CREATE INDEX, ALTER TABLE**
- valor ideal: área que comporte de 75% a 100% do tamanho da maior tabela ou índice do banco de dados
- valores maiores melhoram a performance de vacuum e restauração de dumps
- Default: 16MB
- Sugestão: tamanho da maior relação



- *max_fsm_pages = <num>*
- número de páginas em disco utilizadas pelo mapa de espaços livres (FSM)
- quando uma linha é excluída de uma tabela, ela não é imediatamente removida do disco, mas marcada como “livre” no mapa de espaços livres
- o espaço vago pode então ser reutilizado quando houver uma inclusão de linha
- Sugestão: $\geq 16 * \text{max_fsm_relations}$



- *max_fsm_relations* = *<num>*
- número máximo de relações (tabelas e índices) a serem rastreadas pelo FSM para uso do espaço livre
- Default: 1000
- Sugestão: aumentar para a quantidade total de relações na instância:

```
SELECT count(1) FROM pg_class  
      WHERE relkind in ('r', 'i');
```



- *wal_buffers* = *<num> MB*
- memória usada para os dados do WAL (log de transação do PostgreSQL)
- deve ser suficiente para armazenar a quantia de dados gerados por uma transação típica
- os dados são escritos no disco pelo WAL a cada commit de transação
- Default: 64kB
- Sugestão: 2MB



- *fsync* = *<boolean>*
- assegura que os bancos de dados possam ser recuperados até um estado consistente após uma queda
- quando habilitada (default) as gravações do WAL são imediatamente efetivadas no disco rígido
- se desabilitada, aumenta de forma significativa o desempenho do SGBD
- somente desabilitar em casos especiais!



- *commit_delay* = *<num>*
commit_siblings = *<num>*
- usadas para aumentar o desempenho durante a gravação em disco de múltiplas transações finalizadas de modo concorrente
- se existir um número mínimo de conexões ativas no momento da finalização da transação, o servidor espera por um determinado tempo para gravar diversas transações de uma vez



- *checkpoint_segments* = *<num>*
- número máximo de arquivos de LOG entre checkpoints automáticos do WAL
- quanto mais modificações no BD, mais segmentos são necessários
- Tamanho de cada arquivo: 16MB
- Default: 3
- Sugestão: 64 (~1GB em disco)



- *random_page_cost* = *<num>*
- controla a maneira com as leituras fora de sequência em disco serão tratadas
- um valor mais alto provocará maiores leituras sequenciais (*sequential scans*), indicando a preferência dessas sobre a varredura em índices (*index scans*)
- o valor default deve ser alterado caso os discos sejam mais rápidos



Indicando a efetividade do cache



- *effective_cache_size* = *<num>* MB
- ajuda o otimizador a determinar o quão efetivo o cache em disco do SO é ao decidir se ele deve ou não usar um índice
- valor alto: maiores chances de ocorrer buscas pelo índice (*index scans*)
- valor baixo: pode provocar varreduras sequenciais na tabela (*sequential scans*)
- Default: 128MB
- Sugestão: 50% a 75% da memória RAM



- *autovacuum* = *<boolean>*
- indica se o servidor iniciará o daemon autovacuum (defragmentador)
- Sugestão: true (i.e., deixar ligado)
- *autovacuum_naptime* = *<num>*
- tempo mínimo entre execuções do autovacuum em um banco de dados
- Sugestão: 1min



Opções para o VACUUM



- *autovacuum_vacuum_threshold* = *<num>*
- número mínimo de linhas modificadas ou excluídas de uma tabela para que o comando VACUUM seja disparado nela
- Default: 50, Sugestão: 1000
- *autovacuum_vacuum_scale_factor* = *num*
- fração da tabela para disparar o VACUUM
- Sugestão: 0.2 (20% da tabela)



- *autovacuum_analyze_threshold* = *<num>*
- número mínimo de linhas incluídas, modificadas ou excluídas de uma tabela para que o ANALYZE seja disparado nela
- Default: 50, Sugestão: 250
- *autovacuum_analyze_scale_factor* = *num*
- fração da tabela para disparar o ANALYZE
- Sugestão: 0.1 (10% da tabela)



Introdução aos índices



O que são índices?

- analogia com um livro

texto principal

índice

páginas do livro

folhear o livro

buscar pelo índice

tabela

índice

páginas de dados

varredura sequencial
da tabela

varredura pelo índice



Quais as estratégias?

- antes de tudo: conheça a tua aplicação!
- modo indiscriminado:
 - ponha índice em tudo e depois remova o que for desnecessário
- modo cauteloso:
 - ponha índice nas colunas essenciais e crie mais índices de acordo com a necessidade



Quando indexar?

- em chaves primárias e estrangeiras utilizadas em operações de junção (JOIN)
- quando as chaves são frequentemente usadas em cláusulas WHERE em buscas
- quando existe alta seletividade, isto é, as chaves são bastante restritivas ($< 5\%$)
- quando as chaves são utilizadas em processos de ordenação (ORDER BY)
- quando as chaves são utilizadas em operações de agregação (GROUP BY)



Quando não indexar?

- tabelas muito pequenas
- tabelas atualizadas com frequência
- colunas com pouca variação
- colunas baseadas apenas em desempenho de consultas específicas
- quando o índice não for utilizado em buscas ou ordenações
- índices não utilizados devem ser excluídos!



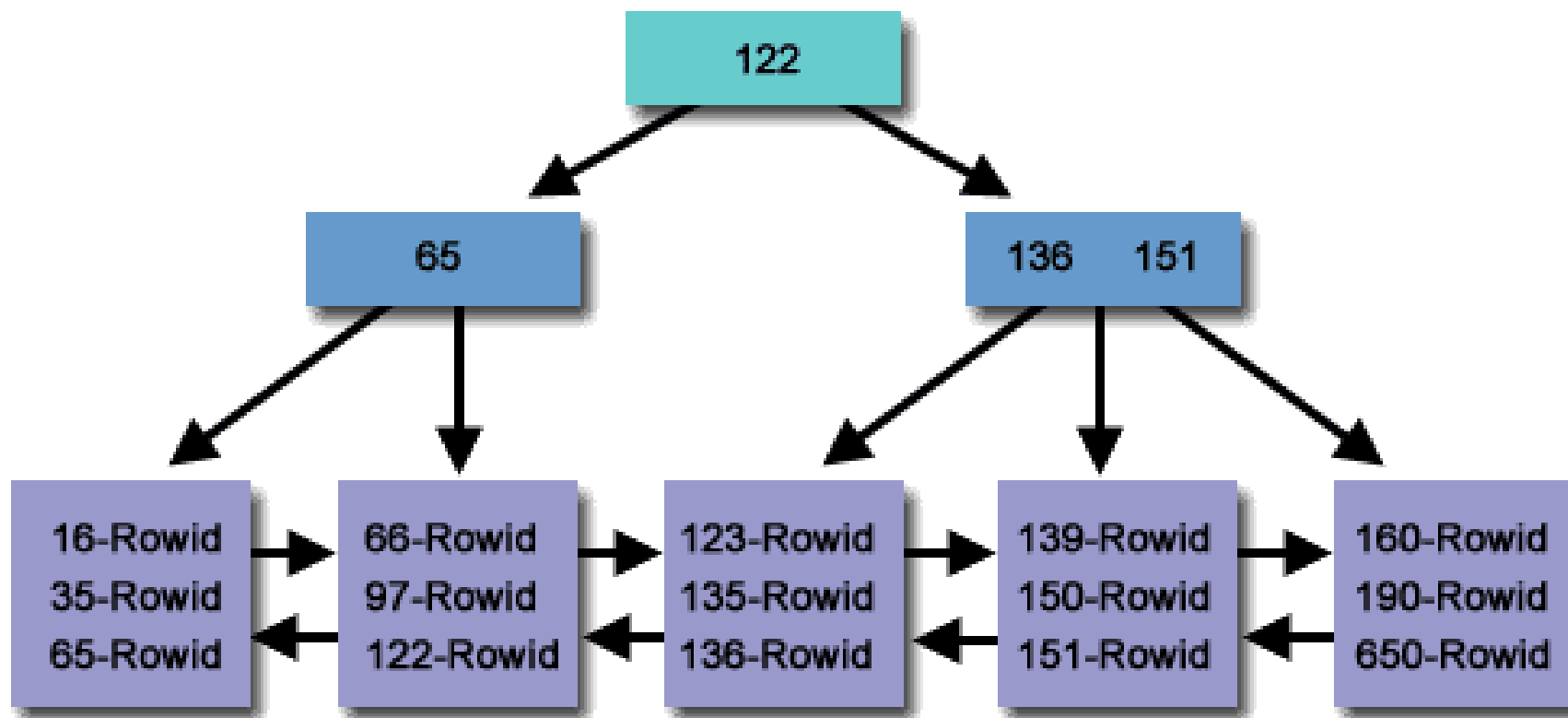
- alto desempenho na busca!

SELECT * FROM produtos WHERE id = 97;

raiz

galho

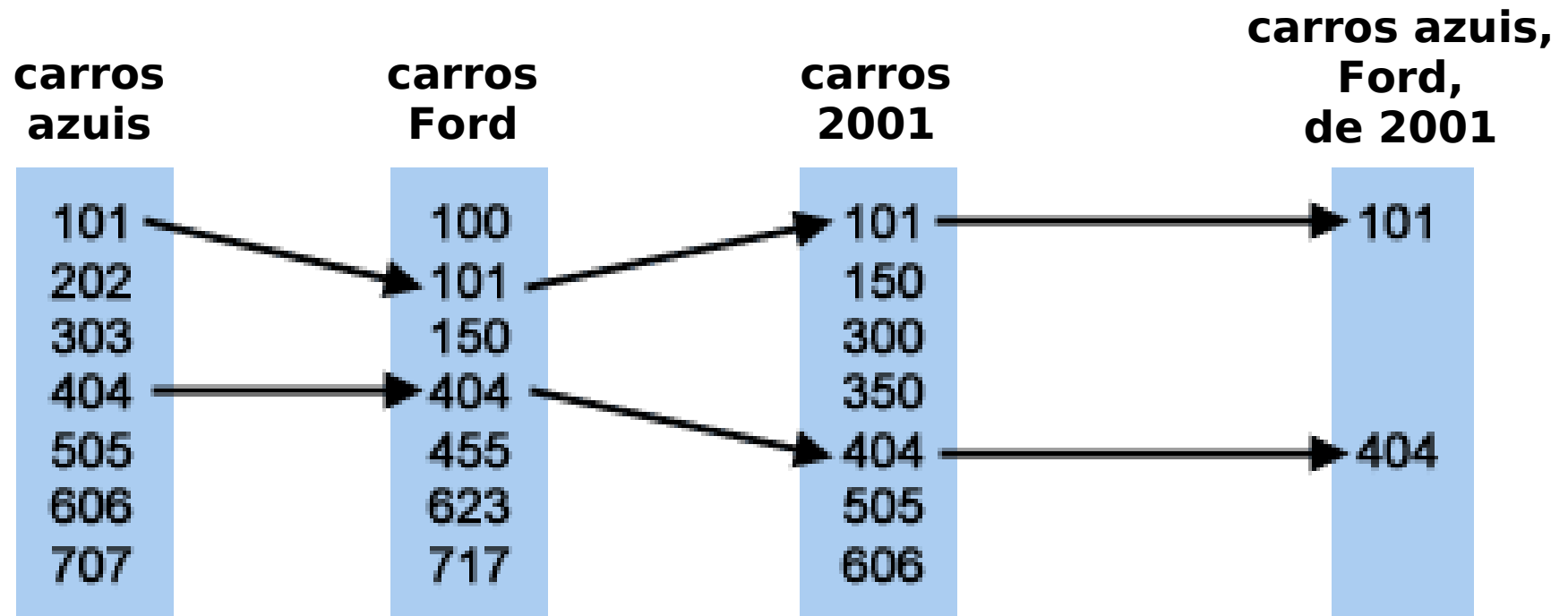
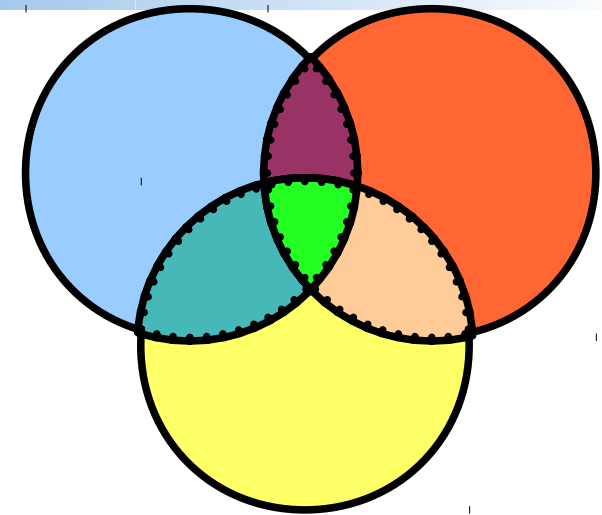
folha





- baixa seletividade individual

```
SELECT * FROM veiculos  
WHERE modelo = 'Ford'  
AND cor = 'azul'  
AND ano = 2001;
```





Utilizando índices no PostgreSQL



- tipos de índices no PostgreSQL:
 - B-tree: implementa árvores B (é o default)
 - Hash: usa tabela de hash para a indexação
 - R-tree: usado em tipos de dados geométricos
 - GiST: usado para tipos de dados customizados
 - GIN: índice invertido
- cada tipo de índice usa um algoritmo distinto adequado para determinados tipos de consultas



Como criar um índice?

```
CREATE [ UNIQUE ] INDEX nome  
  ON tabela [ USING método ]  
    ( { coluna | ( expressão ) } [, ...] )  
  [ TABLESPACE tablespace ]
```

- **UNIQUE**
 - força uma condição de chave única
- **USING**
 - utiliza outro método para a indexação
- **TABLESPACE**
 - o arquivo físico do índice em outra partição



- os índices podem ser criados com base em uma ou mais colunas
- quando o índice múltiplo é utilizado, as primeiras colunas devem sempre ser utilizadas (i.e., problema do “A, B, C”)
- é recomendado que as colunas mais seletivas estejam no início do índice
- exemplo de utilização:

```
CREATE INDEX t1_i01 ON t1 (num, nome);  
CREATE INDEX t1_i02 ON t1 (nome, num);
```

**qual dos índices
será utilizado?**



Como verificar a sua utilização?



- a visão **pg_stat_user_indexes** fornece informações sobre o uso dos índices
- consultando a utilização dos índices

```
SELECT relname, indexrelname, idx_scan,  
        idx_tup_read, idx_tup_fetch  
FROM pg_stat_user_indexes;
```



Como verificar a sua utilização?

- a visão **pg_statio_user_indexes** fornece dados sobre leituras nos blocos dos índices
- consultando a utilização dos índices

```
SELECT relname, indexrelname,  
        idx_blks_read, idx_blks_hit  
FROM pg_statio_user_indexes;
```



A tabela precisa de índices?



- a visão **pg_stat_all_tables** mostra as estatísticas de leituras sequenciais nas tabelas
- consultando a leitura das tabelas

```
SELECT relname, seq_scan, seq_tup_read  
FROM pg_stat_all_tables;
```



- os índices B-tree criam uma estrutura em árvore com a característica de ser balanceada
- comandos de INSERT, UPDATE e DELETE podem desbalancear a árvore
- para mantê-la balanceada, são executadas automaticamente operações de rotação e divisão (*split*), que podem consumir muitos recursos do sistema
- somente mantenha os índices necessários!



```
DROP INDEX [ IF EXISTS ] nome [ , ... ]  
[ CASCADE | RESTRICT ]
```

- **IF EXISTS**
 - caso o índice não exista, não ocorre erro
- **CASCADE**
 - remove também objetos dependentes
- **RESTRICT**
 - não permite remover o índice caso haja dependência de outros objetos



- o PostgreSQL oferece a possibilidade de indexar somente parte da tabela
- esse recurso é extremamente útil para eliminar as linhas com valores mais concentrados e deixar somente os valores mais seletivos
- exemplo de utilização:

```
SELECT * FROM bio WHERE val = 5;
```

```
CREATE INDEX bio_val5 ON bio (val) WHERE val = 5;
```

```
SELECT * FROM bio WHERE val = 5;
```



- qualquer função ou operador sobre uma coluna indexada impede a utilização do índice
- para resolver esse problema, no PostgreSQL um índice pode ser criado com base em um operador ou função
- exemplo de utilização:

```
SELECT * FROM pessoas WHERE substring(nome for 3) = 'MAR';
```

```
CREATE INDEX pessoas_i02  
  ON pessoas (substring(nome for 3));
```



- o PostgreSQL disponibiliza a opção de reordenação física de uma tabela de acordo com um índice existente através do comando CLUSTER
- este recurso é útil para índices com muitas chaves duplicadas e pode ser usado somente para um índice
- o comando CLUSTER deve ser executado periodicamente a fim de manter as linhas da tabela ordenadas com o índice



Otimizando as consultas SQL



- exemplo de utilização do LIKE-%:

```
SELECT * FROM pessoas WHERE nome LIKE 'PAULA %';
```

- para criar índices em campos de texto:

```
CREATE INDEX pessoas_i01  
ON pessoas (nome varchar_pattern_ops);
```

aqui está o truque!

- agora o índice será usado:

```
SELECT * FROM pessoas WHERE nome LIKE 'PAULA %';
```

- desta forma não será usado índice:

```
SELECT * FROM pessoas WHERE nome LIKE '% PAULA';
```



- suponha que uma condição de pesquisa que referencie diversas vezes uma função custosa:

```
... WHERE funcao_custosa(coluna) = 5  
        OR funcao_custosa(coluna) = 15
```

- pode-se utilizar o operador IN para evitar que a função seja invocada diversas vezes na mesma linha:

```
... funcao_custosa(campo) IN (5, 15)
```



- a utilização de operadores em filtros pode fazer com que um índice simplesmente não seja utilizado
- exemplo de caso:

```
SELECT * FROM bio WHERE val = 3 + 2;  
SELECT * FROM bio WHERE val - 2 = 3;
```

**desta forma é
problemático!**



- o PostgreSQL avalia uma condição AND da esquerda para a direita
- colocar a expressão menos provável primeiro pode trazer ganhos
- numa série de expressões OR, coloque a mais provável primeiro



Analizando os planos de acesso



- no PostgreSQL o plano de execução de uma instrução SQL pode ser exibido através do comando EXPLAIN
- o plano de execução mostra:
 - como as tabelas referenciadas pela consulta serão varridas
 - quais algoritmos de junção serão usados para unir as linhas requisitadas de cada tabela
- exemplo de utilização do comando:

```
EXPLAIN SELECT * FROM pessoas WHERE nome LIKE 'ROBERTA%';
```



- a parte mais crítica exibida pelo EXPLAIN é o custo estimado da execução da consulta: a estimativa feita pelo otimizador para o tempo que levará a instrução SQL
- as saídas do EXPLAIN são:
 - custo inicial em operações realizadas antes da recuperação de linhas
 - custo total de execução do comando
 - número de linhas geradas pelo plano
 - comprimento médio dos registros



- a opção ANALYZE do comando EXPLAIN faz com que a consulta seja de fato executada, e não apenas planejada, sendo útil para obter o tempo real de duração
- um dos campos mais importantes da saída do EXPLAIN é o custo final, que deve ser comparado após cada otimização



- o otimizador pode escolher uma das três opções de junção quando a consulta envolve mais de uma tabela:
 - nested loop: consiste na execução de um loop aninhado – a tabela interna deve possuir um índice para ser vantajoso
 - merge join: cada tabela da junção é varrida uma única vez e todas combinadas ao final
 - hash join: tabelas de hash são criadas para cada tabela e depois servem de auxílio no processo de junção



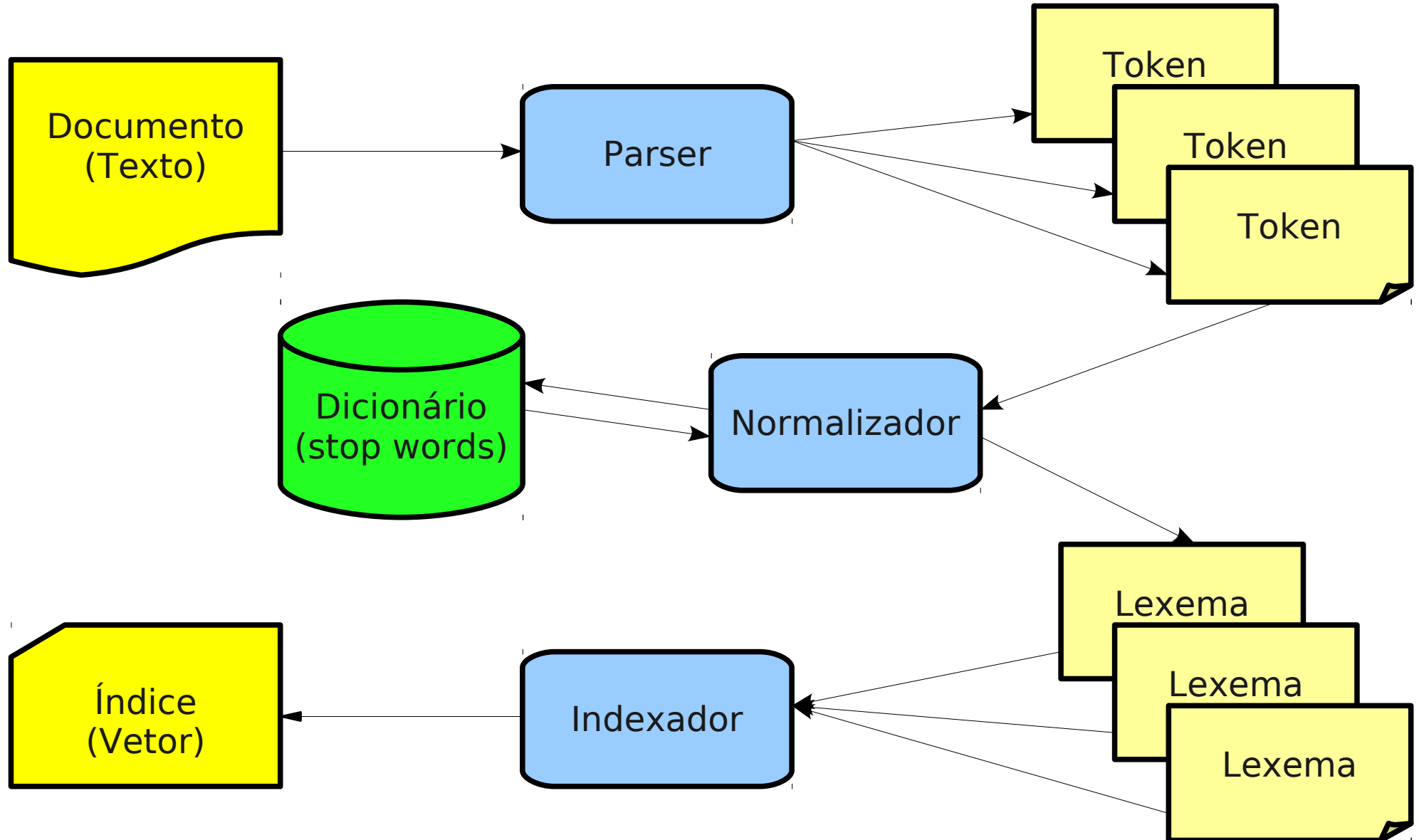
- o plano do otimizador pode ser modificado utilizando os seguintes parâmetros:
 - **enable_seqscan**: se desabilitado, a tabela sempre será consultada por um índice
 - **enable_nestloop**: se desabilitado, não faz junções com a opção nested loop
 - **enable_mergejoin**: se desabilitado, não faz junções com a opção merge join
 - **enable_hashjoin**: se desabilitado, não faz junções com a opção hash join
- pode ser feita em tempo de execução



Buscas textuais (Full Text Search)



Indexação do documento





- criar coluna de busca:

```
ALTER TABLE municipios ADD busca tsvector;
```

- popular coluna de busca:

```
UPDATE municipios  
SET busca = to_tsvector(simples(nome));
```

- efetuar busca usando FTS:

```
SELECT nome, uf FROM municipios  
WHERE busca @@ plainto_tsquery(simples('agua lindaia'));
```

- criar índice especial:

```
CREATE INDEX municipios_gidx  
ON municipios USING gin(busca);
```



- criar função de trigger:

```
CREATE FUNCTION municipios_trigger()  
RETURNS trigger AS $$  
begin  
    new.busca := to_tsvector(simples(new.nome));  
    return new;  
end  
$$ LANGUAGE plpgsql;
```

- criar trigger de atualização:

```
CREATE TRIGGER municipios_tsupdate  
    BEFORE INSERT OR UPDATE ON municipios  
    FOR EACH ROW EXECUTE PROCEDURE municipios_trigger();
```



- buscas com ranking:

```
SELECT nome, uf, ts_rank_cd(busca,  
    to_tsquery('sao & (carlos | pedro)'), 10) as rank  
FROM municipios  
WHERE busca @@ to_tsquery('sao & (carlos | pedro)')  
ORDER BY rank DESC, nome, uf  
LIMIT 10;
```

- depuração dos parsers:

```
SELECT ts_debug('15 contas no http://gmail.com');
```

- estatísticas do vetor de buscas:

```
SELECT * FROM ts_stat('SELECT busca FROM municipios')  
ORDER BY nentry DESC, ndoc DESC, word  
LIMIT 10;
```




Técnicas para carga em massa



- 1. desabilitar auto-commit
 - utilizando BEGIN / COMMIT vários comandos serão executados de uma só vez
- 2. utilizar comando COPY
 - inclusões via INSERT INTO são muito mais lentas pois exigem trabalho de *parsing*
- 3. remover índices
 - árvores dos índices causam *overhead*
- 4. remover chaves estrangeiras
 - restrições não precisam ser validadas



- 5. desabilitar sincronismo do WAL
 - consiste em alterar o parâmetro fsync (não se esqueça de habilitá-lo novamente!)
- 6. aumentar memória para manutenção
 - parâmetro maintenance_work_mem
- 7. aumentar segmentos de LOG
 - parâmetro checkpoint_segments
- 8. atualizar estatísticas ao final
 - ao executar o comando ANALYZE as estatísticas das tabelas serão colhidas



Incluindo linhas com COPY



```
COPY tablename [ ( column [, ...] ) ]  
FROM { 'filename' | STDIN }
```

- altamente recomendável para a inclusão de muitos registros!

- copiando pelo terminal

```
COPY produtos FROM stdin;
```

finalizar com “\.”

- copiando a partir de um arquivo

```
COPY produtos FROM '/tmp/prod.dat';
```

\N	valor nulo
[tab]	separador de campos

caminho absoluto:
o arquivo deve
existir no servidor




```
COPY tablename [ ( column [, ...] ) ]  
TO { 'filename' | STOUT }
```

- útil para a exportação rápida de todas as linhas de uma determinada tabela
- copiando para o terminal

```
COPY produtos TO stdout;
```

- copiando para um arquivo

```
COPY produtos TO '/tmp/prod.dat';
```



**caminho absoluto:
deve haver permissão
de escrita no servidor**



Trabalhando com o formato CSV

- exportando para o formato CSV

```
COPY produtos TO stdout DELIMITER ';' CSV;
```

- importando a partir de um arquivo CSV

```
$ cat > impcsv.sql << EOF  
COPY produtos  
FROM stdin DELIMITER ';' CSV;  
6;Maionese;3.45  
7;Catchup;2.96  
\  
EOF
```

altere o nome
do esquema

não esqueça do
terminador "\."

```
$ psql curso sa_curso < impcsv.sql
```



Bibliografia

PostgreSQL 8.3 Documentation

<http://postgresql.org/docs/8.3/>

Rodrigo Hjort
rodrigo@hjort.co