Estudo de Índices em Geral e no PostgreSQL

CREATE INDEX

Nome

CREATE INDEX -- cria um índice

Sinopse

```
CREATE [ UNIQUE ] INDEX nome_do_índice ON tabela [ USING método ]
    ( { coluna | ( expressão ) } [ classe_de_operadores ] [, ...] )
    [ TABLESPACE espaço_de_tabelas ]
    [ WHERE predicado ]
```

Descrição

O comando CREATE INDEX constrói o índice nome_do_índice na tabela especificada. Os índices são utilizados, principalmente, para melhorar o desempenho do banco de dados (embora a utilização não apropriada possa resultar em uma degradação de desempenho). [1] [2] [3] [4]

Os campos chave para o índice são especificados como nomes de coluna ou, também, como expressões escritas entre parênteses. Podem ser especificados vários campos, se o método de índice suportar índices multicolunas.

O campo de um índice pode ser uma expressão computada a partir dos valores de uma ou mais colunas da linha da tabela. Esta funcionalidade pode ser utilizada para obter acesso rápido aos dados baseado em alguma transformação dos dados básicos. Por exemplo, um índice computado como upper(col) permite a cláusula WHERE upper(col) = 'JIM' utilizar um índice.

O PostgreSQL fornece os métodos de índice B-tree, R-tree, hash e GiST. O método de índice B-tree é uma implementação das *B-trees* de alta concorrência de Lehman-Yao [5] . O método de índice R-tree implementa R-trees padrão utilizando o algoritmo de divisão quadrática (*quadratic split*) de Guttman [6] . O método de índice hash é uma implementação do hash linear de Litwin [7] [8] . Os usuários também podem definir seus próprios métodos de índice, mas é muito complicado.

Quando a cláusula WHERE está presente, é criado um *índice parcial*. Um índice parcial é um índice contendo entradas para apenas uma parte da tabela, geralmente uma parte mais útil para indexar do que o restante da tabela. Por exemplo, havendo uma tabela contendo tanto pedidos faturados quanto não faturados, onde os pedidos não faturados ocupam uma pequena parte da tabela, mas que é bastante usada, o desempenho pode ser melhorado criando um índice apenas para esta parte da tabela. Outra aplicação possível é utilizar a cláusula WHERE junto com UNIQUE para impor a unicidade de um subconjunto dos dados da tabela. Para obter informações adicionais deve ser consultada a Seção 11.7 .

A expressão utilizada na cláusula WHERE pode referenciar apenas as colunas da tabela subjacente, mas pode usar todas as colunas, e não apenas as que estão sendo indexadas. Atualmente não são permitidas subconsultas e expressões de agregação na cláusula WHERE. As mesmas restrições se aplicam aos campos do índice que são expressões.

Todas as funções e operadores utilizados na definição do índice devem ser "imutáveis" (*immutable*), ou seja, seus resultados devem depender somente de seus argumentos, e nunca de uma influência externa (como o conteúdo de outra tabela ou a hora atual). Esta restrição garante que o comportamento do índice é bem definido. Para utilizar uma função definida pelo usuário na expressão do índice ou na cláusula WHERE, a função deve ser marcada como IMMUTABLE na sua criação.

Parâmetros

UNIQUE

Faz o sistema verificar valores duplicados na tabela quando o índice é criado, se existirem dados, e toda vez que forem adicionados dados. A tentativa de inserir ou de atualizar dados que resultem em uma entrada duplicada gera um erro.

nome_do_índice

O nome do índice a ser criado. O nome do esquema não pode ser incluído aqui; o índice é sempre criado no mesmo esquema da tabela que este pertence.

tabela

O nome (opcionalmente qualificado pelo esquema) da tabela a ser indexada.

método

O nome do método de índice a ser utilizado. Pode ser escolhido entre btree, hash, rtree e gist. O método padrão é btree.

coluna

O nome de uma coluna da tabela.

expressão

Uma expressão baseada em uma ou mais colunas da tabela. Geralmente a expressão deve ser escrita entre parênteses, conforme mostrado na sintaxe. Entretanto, os parênteses podem ser omitidos se a expressão tiver a forma de uma chamada de função.

classe_de_operadores

O nome de uma classe de operadores. Veja os detalhes abaixo.

espaço_de_tabelas

O espaço de tabelas onde o índice será criado. Se não for especificado, será utilizado o default_tablespace, ou o espaço de tabelas padrão do banco de dados se default_tablespace for uma cadeia de caracteres vazia.

predicado

A expressão de restrição para o índice parcial.

Observações

Consulte o Capítulo 11 para obter informações sobre quando os índices podem ser utilizados, quando não são utilizados, e em quais situações particulares podem ser úteis.

Atualmente somente os métodos de índice B-tree e GiST suportam índices com mais de uma

coluna. Por padrão podem ser especificadas até 32 colunas (este limite pode ser alterado na construção do PostgreSQL). Também atualmente somente B-tree suporta índices únicos.

Pode ser especificada uma *classe de operadores* para cada coluna de um índice. A classe de operadores identifica os operadores a serem utilizados pelo índice para esta coluna. Por exemplo, um índice B-tree sobre inteiros de quatro bytes usaria a classe int4_ops; esta classe de operadores inclui funções de comparação para inteiros de quatro bytes. Na prática, a classe de operadores padrão para o tipo de dado da coluna é normalmente suficiente. O ponto principal em haver classes de operadores é que, para alguns tipos de dado, pode haver mais de uma ordenação que faça sentido. Por exemplo, pode-se desejar classificar o tipo de dado do número complexo tanto pelo valor absoluto quanto pela parte real, o que pode ser feito definindo duas classes de operadores para o tipo de dado e, então, selecionando a classe apropriada na construção do índice. Mais informações sobre classes de operadores estão na Seção 11.6 e na Seção 31.14.

Para remover um índice deve ser utilizado o comando *DROP INDEX*.

Por padrão não é utilizado índice para a cláusula IS NULL. A melhor forma para utilizar índice nestes casos é a criação de um índice parcial usando o predicado IS NULL.

Exemplos

Para criar um índice B-tree para a coluna titulo na tabela filmes:

```
CREATE UNIQUE INDEX unq_titulo ON filmes (titulo);
```

Para criar um índice para a coluna codigo da tabela filmes e fazer o índice residir no espaço de tabelas espaco_indices:

```
CREATE INDEX idx_codigo ON filmes(codigo) TABLESPACE espaco_indices;
```

Compatibilidade

O comando CREATE INDEX é uma extensão do PostgreSQL à linguagem. O padrão SQL não trata de índices.

Consulte também

ALTER INDEX, *DROP INDEX*

Notas

[1] Oracle — O comando CREATE INDEX é utilizado para criar um índice em: 1) Uma ou mais colunas de uma tabela, de uma tabela particionada, de uma tabela organizada pelo índice, de um agrupamento (*cluster* = objeto de esquema que contém dados de uma ou mais tabelas, todas tendo uma ou mais colunas em comum. O banco de dados Oracle armazena todas as linhas de todas as tabelas que compartilham a mesma chave de agrupamento juntas); 2) Um ou mais atributos de objeto tipado escalar de uma tabela ou de um agrupamento; 3) Uma tabela de armazenamento de tabela aninhada para indexar uma coluna de tabela aninhada. Oracle® Database SQL Reference 10g Release 1 (10.1) Part Number B10759-01 (N. do T.)

[2] Oracle — A *tabela organizada pelo índice* é um tipo especial de tabela que armazena as linhas da tabela dentro de segmento de índice. A tabela organizada pelo índice também pode ter um segmento de estouro (*overflow*) para armazenar as linhas que não cabem no segmento de

índice original. Hands-On Oracle Database 10g Express Edition for Linux — Steve Bobrowski, pág. 350 (N. do T.)

[3] SQL Server — O comando CREATE INDEX cria um índice relacional em uma tabela ou visão especificada, ou um índice XML em uma tabela especificada. O índice pode ser criado antes de existir dado na tabela. Podem ser criados índices para tabelas e visões em outros bancos de dados especificando um nome de banco de dados qualificado. O argumento CLUSTERED cria um índice em que a ordem lógica dos valores da chave determina a ordem física das linhas correspondentes da tabela. A criação de um índice agrupado (*clustered*) único em uma visão melhora o desempenho, porque a visão é armazenada no banco de dados da mesma maneira que a tabela com um índice agrupado é armazenada. Podem ser criados índices em colunas calculadas. No SQL Server 2005 as colunas calculadas podem ter a propriedade PERSISTED. Isto significa que o Mecanismo de Banco de Dados armazena os valores calculados, e os atualiza quando as outras colunas das quais a coluna calculada depende são atualizadas. O Mecanismo de Banco de Dados utiliza os valores persistentes quando cria o índice para a coluna, e quando o índice é referenciado em uma consulta. SQL Server 2005 Books Online — CREATE INDEX (Transact-SQL) (N. do T.)

[4] DB2 — O comando CREATE INDEX é utilizado para: Definir um índice em uma tabela do DB2 (o índice pode ser definido em dados XML ou dados relacionais); Criar uma especificação de índice (metadados que indicam ao otimizador que a tabela de origem possui um índice). A cláusula CLUSTER especifica que o índice é o índice agrupador da tabela. O fator de agrupamento de um índice agrupador é mantido ou melhorado dinamicamente à medida que os dados são inseridos na tabela associada, tentando inserir as novas linhas fisicamente próximas das linhas para as quais os valores chave deste índice estão no mesmo intervalo. DB2 Version 9 for Linux, UNIX, and Windows (N. do T.)

[5] Lehman, Yao 81 - Philip L. Lehman , s. Bing Yao, Efficient locking for concurrent operations on B-trees, ACM Transactions on Database Systems (TODS), v.6 n.4, p.650-670, Dec. 1981 (N. do T.)

[6] Antonin Guttman: R-Trees: A Dynamic Index Structure for Spatial Searching. SIGMOD Conference 1984. (N. do T.)

[7] Litwin, W. Linear hashing: A new tool for file and table addressing. In Proceedings of the 6th Conference on Very Large Databases, (New York, 1980}, 212-223. (N. do T.)

[8] Witold Litwin: Linear Hashing: A new Tool for File and Table Addressing - Summary by: Steve Gribble and Armando Fox. (N. do T.)

Fonte: http://pgdocptbr.sourceforge.net/pg80/sql-createindex.html

Através da Criação de uma Tabela também podemos criar um índice
(é o que ocorre com mais frequência)

CREATE TABLE -- cria uma tabela

Sinopse

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE nome_da_tabela (
  { nome_da_coluna tipo_de_dado [ DEFAULT expressão_padrão ]
[ restrição_de_coluna [ ... ] ]
    | restrição_de_tabela
    | LIKE tabela_ancestral [ { INCLUDING | EXCLUDING } DEFAULTS ] }  [, ... ]
)
[ INHERITS ( tabela_ancestral [, ... ] ) ]
[ WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE espaço_de_tabelas ]

onde restrição_de_coluna é:

[ CONSTRAINT nome_da_restrição ]
{ NOT NULL |
  NULL |
  UNIQUE [ USING INDEX TABLESPACE espaço_de_tabelas ] |
  PRIMARY KEY [ USING INDEX TABLESPACE espaço_de_tabelas ] |
  CHECK (expressão) |
  REFERENCES tabela_referenciada [ ( coluna_referenciada ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE ação ] [ ON UPDATE ação ] }
    [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

e restrição_de_tabela é:

[ CONSTRAINT nome_da_restrição ]
{ UNIQUE ( nome_da_coluna [, ... ] ) [ USING INDEX TABLESPACE
espaço_de_tabelas ] |
  PRIMARY KEY ( nome_da_coluna [, ... ] ) [ USING INDEX TABLESPACE
espaço_de_tabelas ] |
  CHECK ( expressão ) |
  FOREIGN KEY ( nome_da_coluna [, ... ] )
    REFERENCES tabela_referenciada [ ( coluna_referenciada [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE ação ] [ ON UPDATE
ação ] }
    [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

CREATE INDEX

Name

CREATE INDEX -- define a new index

Synopsis

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] name ON table [ USING method ]
    ( { column | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST |
LAST } ] [, ...] )
```

```
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace ]
    [ WHERE predicate ]
```

Obs.: É bom observar a sintaxe das versões dos manuais existente. Veja que na versão 8.0 do manual em português, não existe o parâmetro CONCURRENTLY.

Description

`CREATE INDEX` constructs an index *`name`* on the specified table. Indexes are primarily used to enhance database performance (though inappropriate use can result in slower performance).

The key field(s) for the index are specified as column names, or alternatively as expressions written in parentheses. Multiple fields can be specified if the index method supports multicolumn indexes.

An index field can be an expression computed from the values of one or more columns of the table row. This feature can be used to obtain fast access to data based on some transformation of the basic data. For example, an index computed on `upper(col)` would allow the clause `WHERE upper(col) = 'JIM'` to use an index.

PostgreSQL provides the index methods B-tree, hash, GiST, and GIN. Users can also define their own index methods, but that is fairly complicated.

When the `WHERE` clause is present, a *partial index* is created. A partial index is an index that contains entries for only a portion of a table, usually a portion that is more useful for indexing than the rest of the table. For example, if you have a table that contains both billed and unbilled orders where the unbilled orders take up a small fraction of the total table and yet that is an often used section, you can improve performance by creating an index on just that portion. Another possible application is to use `WHERE` with `UNIQUE` to enforce uniqueness over a subset of a table. See Section 11.8 for more discussion.

The expression used in the `WHERE` clause can refer only to columns of the underlying table, but it can use all columns, not just the ones being indexed. Presently, subqueries and aggregate expressions are also forbidden in `WHERE`. The same restrictions apply to index fields that are expressions.

All functions and operators used in an index definition must be "immutable", that is, their results must depend only on their arguments and never on any outside influence (such as the contents of another table or the current time). This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index expression or `WHERE` clause, remember to mark the function immutable when you create it.

Parameters

`UNIQUE`

Causes the system to check for duplicate values in the table when the index is created (if data already exist) and each time data is added. Attempts to insert or update data which would result in duplicate entries will generate an error.

`CONCURRENTLY`

When this option is used, PostgreSQL will build the index without taking any locks that prevent concurrent inserts, updates, or deletes on the table; whereas a standard index build locks out writes (but not reads) on the table until it's done. There are several caveats to be aware of when using this option — see *Building Indexes Concurrently*.

`name`

The name of the index to be created. No schema name can be included here; the index is always created in the same schema as its parent table.

`table`

The name (possibly schema-qualified) of the table to be indexed.

`method`

The name of the index method to be used. Choices are `btree`, `hash`, `gist`, and `gin`. The default method is `btree`.

`column`

The name of a column of the table.

`expression`

An expression based on one or more columns of the table. The expression usually must be written with surrounding parentheses, as shown in the syntax. However, the parentheses can be omitted if the expression has the form of a function call.

`opclass`

The name of an operator class. See below for details.

`ASC`

Specifies ascending sort order (which is the default).

`DESC`

Specifies descending sort order.

`NULLS FIRST`

Specifies that nulls sort before non-nulls. This is the default when `DESC` is specified.

`NULLS LAST`

Specifies that nulls sort after non-nulls. This is the default when `DESC` is not specified.

`storage_parameter`

The name of an index-method-specific storage parameter. See below for details.

`tablespace`

The tablespace in which to create the index. If not specified, default_tablespace is consulted, or temp_tablespaces for indexes on temporary tables.

`predicate`

The constraint expression for a partial index.

Index Storage Parameters

The `WITH` clause can specify *storage parameters* for indexes. Each index method can have its own set of allowed storage parameters. The built-in index methods all accept a single parameter:

`FILLFACTOR`

The fillfactor for an index is a percentage that determines how full the index method will try to pack index pages. For B-trees, leaf pages are filled to this percentage during initial index build, and also when extending the index at the right (largest key values). If pages subsequently become completely full, they will be split, leading to gradual degradation in the index's efficiency. B-trees use a default fillfactor of 90, but any value from 10 to 100 can be selected. If the table is static then fillfactor 100 is best to minimize the index's physical size, but for heavily updated tables a smaller fillfactor is better to minimize the need for page splits. The other index methods use fillfactor in different but roughly analogous ways; the default fillfactor varies between methods.

Building Indexes Concurrently

Creating an index can interfere with regular operation of a database. Normally PostgreSQL locks the table to be indexed against writes and performs the entire index build with a single scan of the table. Other transactions can still read the table, but if they try to insert, update, or delete rows in the table they will block until the index build is finished. This could have a severe effect if the system is a live production database. Very large tables can take many hours to be indexed, and even for smaller tables, an index build can lock out writers for periods that are unacceptably long for a production system.

PostgreSQL supports building indexes without locking out writes. This method is invoked by specifying the `CONCURRENTLY` option of `CREATE INDEX`. When this option is used, PostgreSQL must perform two scans of the table, and in addition it must wait for all existing transactions that could potentially use the index to terminate. Thus this method requires more total work than a

standard index build and takes significantly longer to complete. However, since it allows normal operations to continue while the index is built, this method is useful for adding new indexes in a production environment. Of course, the extra CPU and I/O load imposed by the index creation might slow other operations.

In a concurrent index build, the index is actually entered into the system catalogs in one transaction, then the two table scans occur in a second and third transaction. If a problem arises while scanning the table, such as a uniqueness violation in a unique index, the `CREATE INDEX` command will fail but leave behind an "invalid" index. This index will be ignored for querying purposes because it might be incomplete; however it will still consume update overhead. The psql `\d` command will mark such an index as `INVALID`:

```
postgres=# \d tab
       Table "public.tab"
 Column |  Type   | Modifiers
--------+---------+-----------
 col    | integer |
Indexes:
    "idx" btree (col) INVALID
```

The recommended recovery method in such cases is to drop the index and try again to perform `CREATE INDEX CONCURRENTLY`. (Another possibility is to rebuild the index with `REINDEX`. However, since `REINDEX` does not support concurrent builds, this option is unlikely to seem attractive.)

Another caveat when building a unique index concurrently is that the uniqueness constraint is already being enforced against other transactions when the second table scan begins. This means that constraint violations could be reported in other queries prior to the index becoming available for use, or even in cases where the index build eventually fails. Also, if a failure does occur in the second scan, the "invalid" index continues to enforce its uniqueness constraint afterwards.

Concurrent builds of expression indexes and partial indexes are supported. Errors occurring in the evaluation of these expressions could cause behavior similar to that described above for unique constraint violations.

Regular index builds permit other regular index builds on the same table to occur in parallel, but only one concurrent index build can occur on a table at a time. In both cases, no other types of schema modification on the table are allowed meanwhile. Another difference is that a regular `CREATE INDEX` command can be performed within a transaction block, but `CREATE INDEX CONCURRENTLY` cannot.

Notes

See Chapter 11 for information about when indexes can be used, when they are not used, and in which particular situations they can be useful.

Currently, only the B-tree and GiST index methods support multicolumn indexes. Up to 32 fields can be specified by default. (This limit can be altered when building PostgreSQL.) Only B-tree currently supports unique indexes.

An *operator class* can be specified for each column of an index. The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on four-byte integers would use the `int4_ops` class; this operator class includes comparison functions for four-byte integers. In practice the default operator class for the column's data type is usually sufficient. The main point of having operator classes is that for some data types, there could be more than one meaningful ordering. For example, we might want to sort a complex-number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type

and then selecting the proper class when making an index. More information about operator classes is in Section 11.9 and in Section 34.14.

For index methods that support ordered scans (currently, only B-tree), the optional clauses `ASC`, `DESC`, `NULLS FIRST`, and/or `NULLS LAST` can be specified to reverse the normal sort direction of the index. Since an ordered index can be scanned either forward or backward, it is not normally useful to create a single-column `DESC` index — that sort ordering is already available with a regular index. The value of these options is that multicolumn indexes can be created that match the sort ordering requested by a mixed-ordering query, such as `SELECT ... ORDER BY x ASC, y DESC`. The `NULLS` options are useful if you need to support "nulls sort low" behavior, rather than the default "nulls sort high", in queries that depend on indexes to avoid sorting steps.

Use *DROP INDEX* to remove an index.

Prior releases of PostgreSQL also had an R-tree index method. This method has been removed because it had no significant advantages over the GiST method. If `USING rtree` is specified, `CREATE INDEX` will interpret it as `USING gist`, to simplify conversion of old databases to GiST.

Examples

To create a B-tree index on the column `title` in the table `films`:

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

To create an index on the expression `lower(title)`, allowing efficient case-insensitive searches:

```
CREATE INDEX lower_title_idx ON films ((lower(title)));
```

To create an index with non-default sort ordering of nulls:

```
CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);
```

To create an index with non-default fill factor:

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH (fillfactor = 70);
```

To create an index on the column `code` in the table `films` and have the index reside in the tablespace `indexspace`:

```
CREATE INDEX code_idx ON films(code) TABLESPACE indexspace;
```

To create an index without locking out writes to the table:

```
CREATE INDEX CONCURRENTLY sales_quantity_index ON sales_table (quantity);
```

Compatibility

`CREATE INDEX` is a PostgreSQL language extension. There are no provisions for indexes in the SQL standard.

See Also

ALTER INDEX, *DROP INDEX*

Fonte: http://www.postgresql.org/docs/8.3/static/sql-createindex.html

Índices parciais

O *índice parcial* é um índice construído sobre um subconjunto da tabela; o subconjunto é definido por uma expressão condicional (chamada de *predicado* [1] do índice parcial). O índice contém entradas apenas para as linhas da tabela que satisfazem o predicado. [2]

O principal motivo para criar índices parciais é evitar a indexação de valores freqüentes. Como um comando procurando por um valor freqüente (um que apareça em mais que uma pequena percentagem de linhas da tabela) não utiliza o índice de qualquer forma, não faz sentido manter estas linhas no índice. Isto reduz o tamanho do índice, acelerando as consultas que utilizam este índice. Também acelera muitas operações de atualização da tabela, porque o índice não precisa ser atualizado em todos os casos. O Exemplo 11-1 mostra uma aplicação possível desta idéia.

Exemplo 11-1. Definir um índice parcial excluindo valores freqüentes

Suponha que os registros de acesso ao servidor *Web* são armazenadas no banco de dados, e que a maioria dos acessos se origina na faixa de endereços de IP da própria organização, mas alguns são de fora (digamos, empregados com acesso discado). Se a procura por endereços de IP for principalmente sobre acesso externo, provavelmente não será necessário indexar a faixa de endereços de IP correspondente à subrede da própria organização.

Assumindo que exista uma tabela como esta:

```
CREATE TABLE tbl_registro_acesso (
    url         varchar,
    ip_cliente  inet,
    ...
);
```

Para criar um índice parcial adequado ao exemplo acima, deve ser utilizado um comando como:

```
CREATE INDEX idx_registro_acesso_ip_cliente ON tbl_registro_acesso (ip_cliente)
    WHERE NOT (ip_cliente > inet '192.168.100.0' AND ip_cliente < inet
'192.168.100.255');
```

Uma consulta típica que pode utilizar este índice é:

```
SELECT * FROM tbl_registro_acesso WHERE url = '/index.html' AND ip_cliente =
inet '212.78.10.32';
```

Uma consulta típica que não pode utilizar este índice é:

```
SELECT * FROM access_log WHERE client_ip = inet '192.168.100.23';
```

Deve ser observado que este tipo de índice parcial requer que os valores comuns sejam determinados a priori. Se a distribuição dos valores for inerente (devido à natureza da aplicação) e estática (não muda com o tempo) não é difícil, mas se os valores freqüentes se devem meramente à carga de dados coincidentes, pode ser necessário bastante trabalho de manutenção.

Outra possibilidade é excluir do índice os valores para os quais o perfil típico das consultas não tenha interesse, conforme mostrado no Exemplo 11-2. Isto resulta nas mesmas vantagens mostradas acima, mas impede o acesso aos valores "que não interessam" por meio deste índice, mesmo se a varredura do índice for vantajosa neste caso. Obviamente, definir índice parcial para este tipo de cenário requer muito cuidado e experimentação.

Exemplo 11-2. Definir um índice parcial excluindo valores que não interessam

Se existir uma tabela contendo tanto pedidos faturados quanto não faturados, onde os pedidos não

faturados representam uma pequena parte da tabela, mas são os mais acessados, é possível melhorar o desempenho criando um índice somente para os pedidos não faturados. O comando para criar o índice deve ser parecido com este:

```
CREATE INDEX idx_pedidos_nao_faturados ON pedidos (num_pedido)
    WHERE faturado is not true;
```

Uma possível consulta utilizando este índice é

```
SELECT * FROM pedidos WHERE faturado is not true AND num_pedido < 10000;
```

Entretanto, o índice também pode ser utilizado em consultas não envolvendo num_pedido como, por exemplo,

```
SELECT * FROM pedidos WHERE faturado is not true AND valor > 5000.00;
```

Embora não seja tão eficiente quanto seria um índice parcial na coluna valor, porque o sistema precisa percorrer o índice por inteiro, mesmo assim, havendo poucos pedidos não faturados, a utilização do índice parcial para localizar apenas os pedidos não faturados pode ser vantajosa.

Deve ser observado que a consulta abaixo não pode utilizar este índice:

```
SELECT * FROM pedidos WHERE num_pedido = 3501;
```

O pedido número 3501 pode estar entre os pedidos faturados e os não faturados.

O Exemplo 11-2 também ilustra que a coluna indexada e a coluna utilizada no predicado não precisam corresponder. O PostgreSQL suporta índices parciais com predicados arbitrários, desde que somente estejam envolvidas colunas da tabela indexada. Entretanto, deve-se ter em mente que o predicado deve corresponder às condições utilizadas nos comandos que supostamente vão se beneficiar do índice. Para ser preciso, o índice parcial somente pode ser utilizado em um comando se o sistema puder reconhecer que a condição WHERE do comando implica matematicamente no predicado do índice. O PostgreSQL não possui um provador de teoremas sofisticado que possa reconhecer expressões equivalentes matematicamente escritas de forma diferente (Não seria apenas extremamente difícil criar este provador de teoremas geral, como este provavelmente também seria muito lento para poder ser usado na prática). O sistema pode reconhecer implicações de desigualdades simples como, por exemplo, "x < 1" implica "x < 2"; senão, a condição do predicado deve corresponder exatamente a uma parte da condição WHERE da consulta, ou o índice não será reconhecido como utilizável.

Um terceiro uso possível para índices parciais não requer que o índice seja utilizado em nenhum comando. A idéia é criar um índice único sobre um subconjunto da tabela, como no Exemplo 11-3, impondo a unicidade das linhas que satisfazem o predicado do índice, sem restringir as que não fazem parte.

Exemplo 11-3. Definir um índice único parcial

Suponha que exista uma tabela contendo perguntas e respostas. Deseja-se garantir que exista apenas uma resposta "correta" para uma dada pergunta, mas que possa haver qualquer número de respostas "incorretas". Abaixo está mostrada a forma de fazer:

```
CREATE TABLE tbl_teste
    (pergunta  text,
     resposta  text,
     correto   bool
     ...
);

CREATE UNIQUE INDEX unq_resposta_correta ON tbl_teste (pergunta, correto)
    WHERE correto;
```

Esta forma é particularmente eficiente quando existem poucas respostas corretas, e muitas incorretas.

Finalizando, também pode ser utilizado um índice parcial para mudar a escolha do plano de comando feito pelo sistema. Pode ocorrer que conjuntos de dados com distribuições peculiares façam o sistema utilizar um índice quando na realidade não deveria. Neste caso, o índice pode ser definido de modo que não esteja disponível para o comando com problema. Normalmente, o PostgreSQL realiza escolhas razoáveis com relação à utilização dos índices (por exemplo, evita-os ao buscar valores com muitas ocorrências, desta maneira o primeiro exemplo realmente economiza apenas o tamanho do índice, mas não é necessário para evitar a utilização do índice), e a escolha de um plano grosseiramente incorreto é motivo para um relatório de erro.

Deve-se ter em mente que a criação de um índice parcial indica que você sabe pelo menos tanto quanto o planejador de comandos sabe. Em particular, você sabe quando um índice poderá ser vantajoso. A formação deste conhecimento requer experiência e compreensão sobre como os índices funcionam no PostgreSQL. Na maioria dos casos, a vantagem de um índice parcial sobre um índice regular não é muita.

Podem ser obtidas informações adicionais sobre índices parciais em *The case for partial indexes* , *Partial indexing in POSTGRES: research project* e *Generalized Partial Indexes* .

Notas

[1] predicado — especifica uma condição que pode ser avaliada para obter um resultado booleano. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)

[2] Os sistemas gerenciadores de banco de dados SQL Server 2000, Oracle 10g e DB2 8.1 não possuem suporte a índices parciais. Comparison of relational database management systems (N. do T.)

Examinar a utilização do índice

Embora no PostgreSQL os índices não necessitem de manutenção e ajuste, ainda assim é importante verificar quais índices são utilizados realmente pelos comandos executados no ambiente de produção. O exame da utilização de um índice por um determinado comando é feito por meio do comando *EXPLAIN*; sua aplicação para esta finalidade está ilustrada na Seção 13.1. Também é possível coletar estatísticas gerais sobre a utilização dos índices por um servidor em operação da maneira descrita na Seção 23.2.

É difícil formular um procedimento genérico para determinar quais índices devem ser definidos. Existem vários casos típicos que foram mostrados nos exemplos das seções anteriores. Muita verificação experimental é necessária na maioria dos casos. O restante desta seção dá algumas dicas.

- O comando *ANALYZE* sempre deve ser executado primeiro. Este comando coleta estatísticas sobre a distribuição dos valores na tabela. Esta informação é necessária para estimar o número de linhas retornadas pela consulta, que é uma necessidade do planejador para atribuir custos dentro da realidade para cada plano de comando possível. Na ausência de estatísticas reais, são assumidos alguns valores padrão, quase sempre imprecisos. O exame da utilização do índice pelo aplicativo sem a execução prévia do comando ANALYZE é,

portanto, uma causa perdida.

- Devem ser usados dados reais para a verificação experimental. O uso de dados de teste para definir índices diz quais índices são necessários para os dados de teste, e nada além disso. É especialmente fatal utilizar conjuntos de dados de teste muito pequenos. Enquanto selecionar 1.000 de cada 100.000 linhas pode ser um candidato para um índice, selecionar 1 de cada 100 linhas dificilmente será, porque as 100 linhas provavelmente cabem dentro de uma única página do disco, e não existe nenhum plano melhor que uma busca seqüencial em uma página do disco.
  Também deve ser tomado cuidado ao produzir os dados de teste, geralmente não disponíveis quando o aplicativo ainda não se encontra em produção. Valores muito semelhantes, completamente aleatórios, ou inseridos ordenadamente, distorcem as estatísticas em relação à distribuição que os dados reais devem ter.
- Quando os índices não são usados, pode ser útil como teste forçar sua utilização. Existem parâmetros em tempo de execução que podem desativar vários tipos de planos (descritos no Seção 16.4). Por exemplo, desativar varreduras seqüenciais (enable_seqscan) e junções de laço-aninhado (enable_nestloop), que são os planos mais básicos, forçam o sistema a utilizar um plano diferente. Se o sistema ainda assim escolher a varredura seqüencial ou a junção de laço-aninhado então existe, provavelmente, algum problema mais fundamental devido ao qual o índice não está sendo utilizado como, por exemplo, a condição da consulta não corresponde ao índice (Qual tipo de consulta pode utilizar qual tipo de índice é explicado nas seções anteriores).
- Se forçar a utilização do índice não faz o índice ser usado, então existem duas possibilidades: ou o sistema está correto e realmente a utilização do índice não é apropriada, ou a estimativa de custo dos planos de comando não estão refletindo a realidade. Portanto, deve ser medido o tempo da consulta com e sem índices. O comando EXPLAIN ANALYZE pode ser útil neste caso.
- Se for descoberto que as estimativas de custo estão erradas existem, novamente, duas possibilidades. O custo total é calculado a partir do custo por linha de cada nó do plano vezes a seletividade estimada do nó do plano. Os custos dos nós do plano podem ser ajustados usando parâmetros em tempo de execução (descritos no Seção 16.4). A estimativa imprecisa da seletividade é devida a estatísticas insuficientes. É possível melhorar esta situação ajustando os parâmetros de captura de estatísticas (consulte o comando *ALTER TABLE*).
  Se não for obtido sucesso no ajuste dos custos para ficarem mais apropriados, então pode ser necessário o recurso de forçar a utilização do índice explicitamente. Pode-se, também, desejar fazer contato com os desenvolvedores do PostgreSQL para examinar este problema.

Índices com várias colunas

Pode ser definido um índice contendo mais de uma coluna. Por exemplo, se existir uma tabela como:

```
CREATE TABLE teste2 (
  principal  int,
  secundario int,
  nome       varchar
);
```

(Digamos que seja armazenado no banco de dados o diretório `/dev`...) e freqüentemente sejam feitas consultas como

```
SELECT nome
    FROM  teste2
```

```
    WHERE principal = constante AND secundario = constante;
```

então é apropriado definir um índice contendo as colunas principal e secundario como, por exemplo,

```
CREATE INDEX idx_teste2_princ_sec ON teste2 (principal, secundario);
```

Atualmente, somente as implementações de B-tree e GiST suportam índices com várias colunas. Podem ser especificadas até 32 colunas (Este limite pode ser alterado durante a geração do PostgreSQL; consulte o arquivo `pg_config_manual.h`).

O planejador de comandos pode utilizar um índice com várias colunas, para comandos envolvendo a coluna mais à esquerda na definição do índice mais qualquer número de colunas listadas à sua direita, sem omissões. Por exemplo, um índice contendo (a, b, c) pode ser utilizado em comandos envolvendo todas as colunas a, b e c, ou em comandos envolvendo a e b, ou em comandos envolvendo apenas a, mas não em outras combinações (Em um comando envolvendo a e c, o planejador pode decidir utilizar o índice apenas para a, tratando c como uma coluna comum não indexada). Obviamente, cada coluna deve ser usada com os operadores apropriados para o tipo do índice; as cláusulas envolvendo outros operadores não são consideradas.

Os índices com várias colunas só podem ser utilizados se as cláusulas envolvendo as colunas indexadas forem ligadas por AND. Por exemplo,

```
SELECT nome
    FROM  teste2
    WHERE principal = constante OR secundario = constante;
```

não pode utilizar o índice idx_teste2_princ_sec definido acima para procurar pelas duas colunas (Entretanto, pode ser utilizado para procurar apenas a coluna principal).

Os índices com várias colunas devem ser usados com moderação. Na maioria das vezes, um índice contendo apenas uma coluna é suficiente, economizando espaço e tempo. Um índice com mais de três colunas é quase certo não ser útil, a menos que a utilização da tabela seja muito peculiar.

Índices em expressões

Uma coluna do índice não precisa ser apenas uma coluna da tabela subjacente, pode ser uma função ou uma expressão escalar computada a partir de uma ou mais colunas da tabela. Esta funcionalidade é útil para obter acesso rápido às tabelas com base em resultados de cálculos. [1]

Por exemplo, uma forma habitual de fazer comparações não diferenciando letras maiúsculas de minúsculas é utilizar a função lower:

```
SELECT * FROM teste1 WHERE lower(col1) = 'valor';

-- Para não diferenciar maiúsculas e minúsculas, acentuadas ou não (N. do T.)

SELECT * FROM teste1 WHERE lower(to_ascii(col1)) = 'valor';
```

Esta consulta pode utilizar um índice, caso algum tenha sido definido sobre o resultado da operação lower(col1):

```
CREATE INDEX idx_teste1_lower_col1 ON teste1 (lower(col1));

-- Para incluir as letras acentuadas (N. do T.)

CREATE INDEX idx_teste1_lower_ascii_col1 ON teste1 (lower(to_ascii(col1)));
```

Se o índice for declarado como UNIQUE, este impede a criação de linhas cujos valores de col1 diferem apenas em letras maiúsculas e minúsculas, assim como a criação de linhas cujos valores de col1 são realmente idênticos. Portanto, podem ser utilizados índices em expressões para impor restrições que não podem ser definidas como restrições simples de unicidade.

Como outro exemplo, quando são feitas habitualmente consultas do tipo

```
SELECT * FROM pessoas WHERE (primeiro_nome || ' ' || ultimo_nome) = 'Manoel
Silva';
```

então vale a pena criar um índice como:

```
CREATE INDEX idx_pessoas_nome ON pessoas ((primeiro_nome || ' ' ||
ultimo_nome));
```

A sintaxe do comando CREATE INDEX normalmente requer que se escreva parênteses em torno da expressão do índice, conforme mostrado no segundo exemplo. Os parênteses podem ser omitidos quando a expressão for apenas uma chamada de função, como no primeiro exemplo.

É relativamente dispendioso manter expressões de índice, uma vez que a expressão derivada deve ser computada para cada linha inserida, ou sempre que for atualizada. Portanto, devem ser utilizadas somente quando as consultas que usam o índice são muito freqüentes.

Notas

[1] O sistema de gerenciamento de banco de dados Oracle 10g também permite usar função e expressão escalar na coluna do índice, mas o SQL Server 2000 e o DB2 8.1 não permitem. Comparison of relational database management systems (N. do T.)

Remoção dos índices

Se estiver sendo carregada uma tabela recém criada, a maneira mais rápida é criar a tabela, carregar os dados usando o COPY e, depois, criar os índices necessários para a tabela. Criar um índice sobre dados pré-existentes é mais rápido que atualizá-lo de forma incremental durante a carga de cada linha.

Para aumentar uma tabela existente, pode-se remover o índice, carregar a tabela e, depois, recriar o índice. É claro que o desempenho do banco de dados para os outros usuários será afetado negativamente durante o tempo que o índice não existir. Deve-se pensar duas vezes antes de remover um índice único, porque a verificação de erro efetuada pela restrição de unicidade não existirá enquanto o índice não tiver sido criado novamente.

ALTER INDEX

Nome

ALTER INDEX -- altera a definição de um índice

Sinopse

```
ALTER INDEX nome
    ação [, ... ]
ALTER INDEX nome
    RENAME TO novo_nome

onde ação é um entre:

    OWNER TO novo_dono
    SET TABLESPACE nome_do_espaço_de_índices
```

Descrição

O comando ALTER INDEX altera a definição de um índice existente. Existem diversas subformas:

OWNER

> Esta forma torna o usuário especificado o dono do índice. Somente pode ser utilizado por um superusuário.

SET TABLESPACE

> Esta forma altera o espaço de tabelas do índice para o espaço de tabelas especificado, e move os arquivos de dados associados ao índice para o novo espaço de tabelas. Consulte também *CREATE TABLESPACE*.

RENAME

> A forma RENAME muda o nome do índice. Não há efeito sobre os dados armazenados.

Todas as ações, com exceção de RENAME, podem ser combinadas em uma lista de alterações múltiplas a serem aplicadas em paralelo.

Parâmetros

nome

> O nome (opcionalmente qualificado pelo esquema) de um índice existente.

novo_nome

> O novo nome do índice.

novo_dono

O nome de usuário do novo dono do índice.

nome_do_espaço_de_tabelas

O nome do espaço de tabelas para o qual o índice será movido.

Observações

Estas operações também podem ser feitas utilizando *ALTER TABLE*. O comando ALTER INDEX é, na verdade, apenas um sinônimo para as formas de ALTER TABLE que se aplicam aos índices.

Não é permitido alterar qualquer parte de um índice dos catálogos do sistema.

Exemplos

Para mudar o nome de um índice existente:

```
ALTER INDEX distribuidores RENAME TO fornecedores;
```

Para mover um índice para outro espaço de tabelas:

```
ALTER INDEX distribuidores SET TABLESPACE espaco_de_tabelas_rapido;
```

Compatibilidade

O comando ALTER INDEX é uma extensão do PostgreSQL.

Mudar Dono de Índice

Para mudar o dono de uma tabela, índice, seqüência ou visão deve ser utilizado o comando ALTER TABLE.

Índices únicos

Os índices também podem ser utilizados para impor a unicidade do valor de uma coluna, ou a unicidade dos valores combinados de mais de uma coluna.

```
CREATE UNIQUE INDEX nome ON tabela (coluna [, ...]);
```

Atualmente, somente os índices B-tree poder ser declarados como únicos.

Quando o índice é declarado como único, não pode existir na tabela mais de uma linha com valores indexados iguais. Os valores nulos não são considerados iguais. Um índice único com várias colunas rejeita apenas os casos onde todas as colunas indexadas são iguais em duas linhas.

O PostgreSQL cria, automaticamente, um índice único quando é definida na tabela uma restrição de unicidade ou uma chave primária. O índice abrange as colunas que compõem a chave primária ou as colunas únicas (um índice com várias colunas, se for apropriado), sendo este o mecanismo que

impõe a restrição.

> **Nota:** A forma preferida para adicionar restrição de unicidade a uma tabela é por meio do comando ALTER TABLE ... ADD CONSTRAINT. A utilização de índices para impor restrições de unicidade pode ser considerada um detalhe de implementação que não deve ser acessado diretamente. Entretanto, deve-se ter em mente que não é necessário criar índices em colunas únicas manualmente; caso se faça, simplesmente se duplicará o índice criado automaticamente.

DROP INDEX

Nome

DROP INDEX -- remove um índice

Sinopse

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

Descrição

O comando DROP INDEX remove do sistema de banco de dados um índice existente. Para executar este comando é necessário ser o dono do índice.

Parâmetros

nome

O nome (opcionalmente qualificado pelo esquema) do índice a ser removido.

CASCADE

Remove automaticamente os objetos que dependem do índice.

RESTRICT

Recusa remover o índice se existirem objetos que dependem do mesmo. Este é o padrão.

Exemplos

O comando a seguir remove o índice idx_titulo:

```
DROP INDEX idx_titulo;
```

Compatibilidade

O comando DROP INDEX é uma extensão do PostgreSQL à linguagem. O padrão SQL não trata de índices.

Índices – de Daniel Oslei

Olá pessoal. Desculpem pela demora na edição de um novo artigo para a Coluna PostgreSQL iMasters.

No primeiro artigo, anunciei que a Coluna PostgreSQL não tratará apenas de assuntos específicos desse SGBD, mas também de conceitos de bancos de dados, e é isso que ocorrerá neste primeiro artigo sobre índices. PostgreSQL ficará um pouco de lado, para que tratemos de uma breve introdução sobre o que são índices. No próximo artigo trataremos mais especificamente os índices no PostgreSQL.

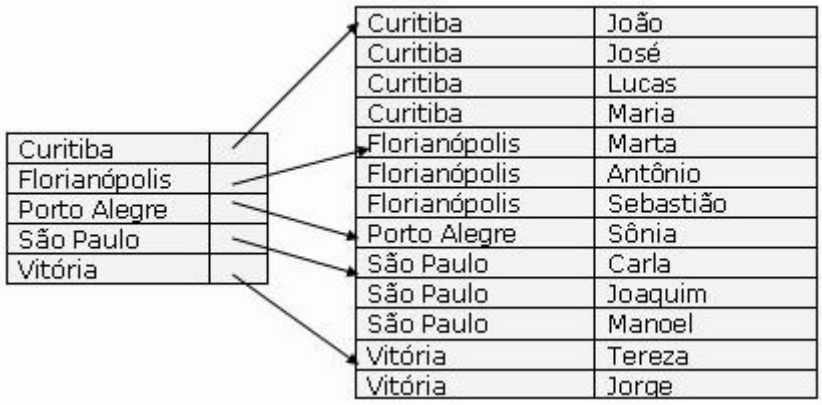
Em grande parte das consultas que são feitas a uma base de dados, fazem referência a apenas uma pequena proporção dos registros de um arquivo. Por exemplo, numa base de dados de uma grande empresa podem ser feitas muito mais consultas para se saber os clientes que moram na cidade de Curitiba do que consultas para saber quem são os clientes de Porto Alegre. E dentro das pesquisas dos clientes curitibanos, podem haver muito mais pesquisas em relação a quem mora em um determinado bairro do que em outros. Esses são exemplos bem simples de como determinadas consultas são feitas repetidamente e várias vezes ao dia. Mas imaginem se o SGBD para fazer essas consultas tenha que analisar registro por registro para poder retornar um resultado. Se a quantidade de registros for pouca, isso será imperceptível. Mas se pelo contrário, existirem um enorme número de linhas em várias tabelas, com várias chaves estrangeiras, começaram a surgir graves problemas de desempenho no sistema. Por isso, é necessária a criação de estruturas para que as consultas sejam executadas na melhor performance possível.

Muitas vezes, quando consultamos um livro, não podemos lê-lo todo para encontrarmos o que procuramos. Se for deste modo há um tempo muito grande sendo desperdiçado. Para isso que existe nos livro os índices, no qual podemos encontrar com mais facilidade o que desejamos. Da mesma maneira, os bancos de dados utilizam índices, para que não só consultas, mas inserções, exclusões e atualizações sejam feitas com mais agilidade.

Habitualmente, os índices são utilizados para melhorar o desempenho dos bancos de dados. Um índice permite ao servidor de banco de dados encontrar e trazer linhas específicas com muito mais rapidez do que faria sem o índice. Mas os índices também produzem trabalho adicional para o sistema de banco de dados como um todo, portanto, deve-se adquirir bons conhecimentos sobre o assunto para o seu devido uso.

Os índices podem beneficiar as atualizações e as exclusões com condição de procura. Eles também podem ser utilizados em consultas com junção. Portanto, um índice definido em uma coluna que faça parte da condição de junção pode acelerar, significativamente, a consulta.

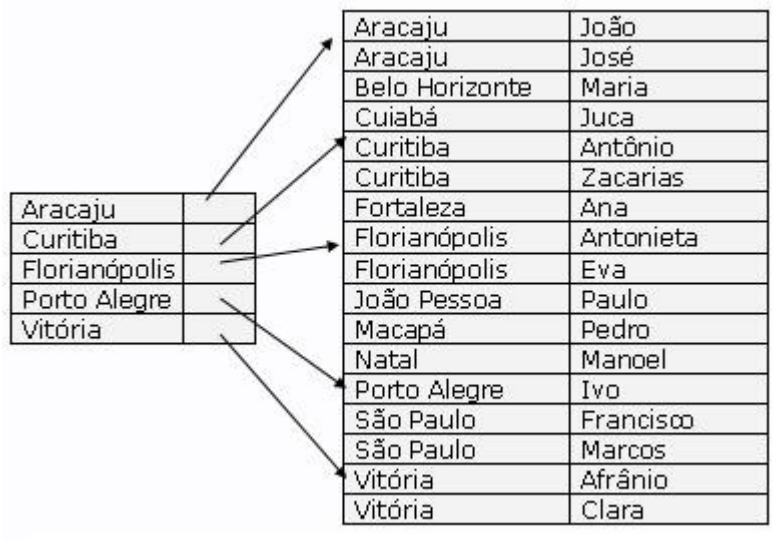
Um dos mais antigos esquemas de índice utilizados em sistema de banco de dados é chamado de arquivo indexado seqüencialmente, que são projetados para aplicações que requerem tanto o processamento seqüencial de um arquivo inteiro quanto o acesso aleatório a registros individuais. Num exemplo um pouco mais avançado, mostramos na imagem abaixo uma tabela na qual os registros são indexados pelo nome da cidade em que moram os clientes. Para encontrarmos os clientes de uma determinada cidade, encontramos a cidade na primeira tabela e seguimos para onde o ponteiro correspondente está apontando, lendo seqüencialmente até encontrar uma cidade diferente da solicitada:



Índices Densos

Reparem que neste exemplo, para cada cidade existe um registro de índice (ou entrada de índice), mesmo que haja apenas um registro para determinada cidade. Este tipo de colocação de índices é chamado de índices densos. Existe uma outra forma de índice conhecida como índices esparsos.

Veja o exemplo abaixo:

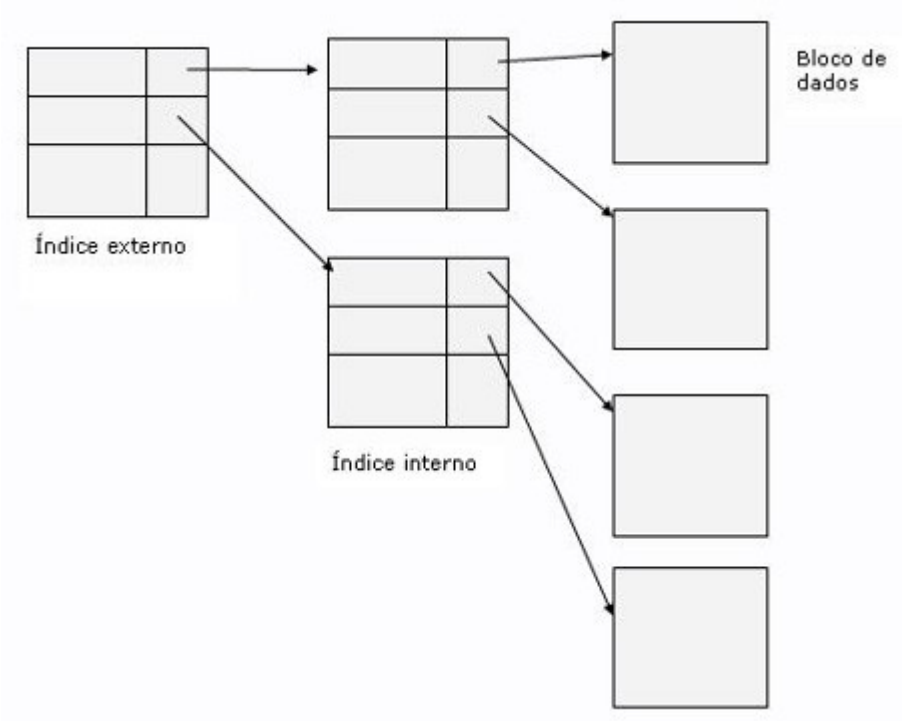


Índices Esparsos

Neste exemplo acima, são criados registros de índices para apenas alguns dos valores. Para localizar um registro, encontramos a entrada de índice com o maior valor de chave de procura que seja menor ou igual ao valor de chave de procura que estamos procurando. Iniciamos no registro apontando para a entrada de índice e seguimos os ponteiros no arquivo até encontrarmos o registro desejado. **Os índices densos são preferíveis comparados aos índices esparsos, devido a possibilidade de encontrarmos com mais agilidade o desejado.** A vantagem dos índices esparsos é o fato de ocuparem pouco espaço em disco e menos trabalho na manutenção em inserções e exclusões.

Mesmo assim, se pensarmos em grandes fontes de armazenamento de dados, essa forma de índices tornaria o desempenho do banco extremamente baixo. Se um índice for suficientemente pequeno para ser mantido na memória principal, o tempo de busca para encontrar uma entrada será baixo. Entretanto, se o índice for tão grande que tenha de ser mantido em disco, a busca de uma entrada exigirá diversas leituras de blocos. Para solucionar este problema, o índice deve ser tratado como qualquer outro arquivo seqüencial, e construímos um índice esparso no índice primário:

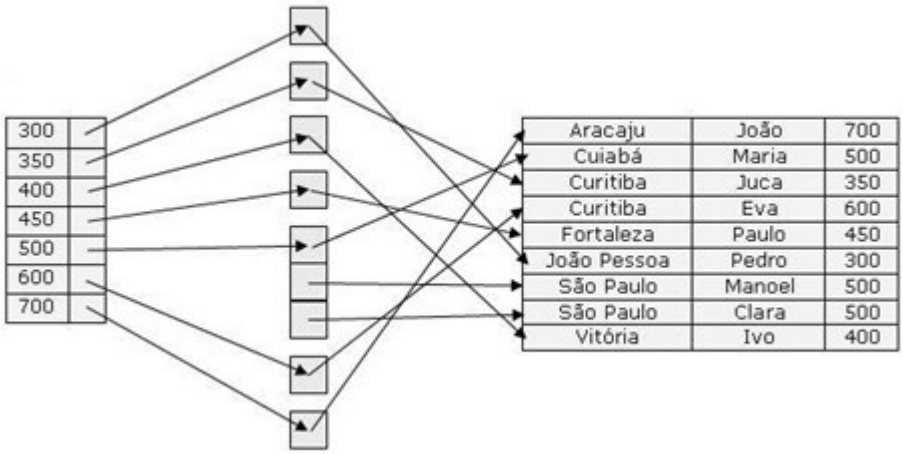
Cada vez que ocorre uma inserção ou remoção de dados, os índices devem ser atualizados. Quando é inserida uma informação, se o índice for denso, é feito uma procura com o dado chave para o índice, se caso não for encontrado, esse valor é incluído no índice. Se o índice for esparso e armazenar uma entrada para cada bloco, não é necessário fazer nenhuma alteração no índice, exceto se um novo bloco tenha sido criado, então o primeiro valor de procura que aparece no novo bloco é inserido. Quando um registro é removido, se o registro for o único para o valor chave de procura, então esse valor chave é excluído do índice. Com índices esparsos, removemos uma valor de chave



e substituímos sua entrada (se houver) no índice pelo próximo valor de chave de procura (na ordem da chave de procura). Se o próximo valor de chave de procura já tiver uma entrada de índice, a entrada é apagada em vez de ser substituída.

São chamados de índices primários os índices que pertencem a uma chave primária ou a que definam a seqüência dos registros. Existem também os índices secundários, que normalmente pertencem a chaves candidatas. Os índices secundários são muito semelhantes aos índices densos, a não ser pelo fato de os registros apontados por valores sucessivos no índice não estão armazenados seqüencialmente, pois, os registros estão dispostos de tal forma a satisfazer a ordenação do índice primário, o que resulta no fato de os índices secundários terem que possuir ponteiros para todos os registros. Uma leitura seqüencial na ordem do índice primário é satisfatório, pois, os registros estão fisicamente armazenados na mesma ordem do índice primário, o que não acontece com os índices secundários. Como a ordem do índice primário e do secundário diferem, provalmente surgiriam complicações com a leitura através da ordem do índice secundário.

Com índices secundários, os ponteiros não apontam diretamente para o arquivo com registros, mas para um *bucket* que contém ponteiros para o arquivo. Veja o exemplo abaixo:



O que foi mostrado acima, é apenas o príncipio de como os índices são feitos, para que servem e como funcionam. Existem algortimos muito mais avançados e complexos do que os já citados, e ainda existem muitos estudos em cima deste assunto, sempre na tentativa de fazer com que os índices sejam mais eficientes, exigindo menos processamento, otimização do espaço em disco e exijam menos manutenção.

O PostgreSQL atualmente implementa quatro tipos de índices: B-Tree, R-Tree, GiST e Hash. Cada um com o seu grau de eficiência, podendo ser recomendado algum deles para uma determinada aplicação e outros para outros tipos de aplicações. Veremos na parte II de nosso artigo, como funciona cada um deles e para o que são recomendados. Na parte III, veremos detalhadamente como são usados.

Indices - Parte 02

Como visto na coluna passada os índices são objetos de vital importância para o bom desempenho do banco de dados. Também foi visto que o PostgreSQL usa índices para melhor responder as requisições solicitadas a ele. Mas tem um porém, o PostgreSQL não cria por conta própria os índices, quem deve criar os índices é o usuário proprietário da tabela, exceto quando é criada uma chave primária, na qual toda chave primária é um índice. E para isso ele já deve ter em mente qual das colunas será mais usada em cláusulas WHERE durante a existência da tabela. Índices são criados usando o comando CREATE INDEX, veja o exemplo:

```
CREATE INDEX nome_indice ON tabela_nome (coluna_nome);
```

Ao criar a chave primária devemos ter em mente qual o campo a ser utilizado na cláusula WHERE.

No exemplo citado acima, foi criado um índice *nome_indice* na coluna *coluna_nome* da tabela *tabela_nome*. Quando o índice é criado com sucesso, é retornado para o prompt a palavra CREATE (é aconselhável utilizar nomes significativos para os índices).

Quando se cria um índice, este índice está relacionado a uma determinada coluna, portanto, não pode auxiliar no acesso de informações em outras colunas porque os índices são classificados de acordo com a coluna correspondente, êis o motivo para o qual deve-se saber muito bem a

conceituação do assunto para criar índices que tragam realmente melhora de desempenho. É claro que você pode criar vários índices dentro de uma mesma tabela, mas um índice que seja raramente usado é um desperdício de espaço em disco e também vários índices requerem que para cada atualização num registro seja também feito uma atualização em cada índice.

Por padrão quando criamos um índice e não especificamos qual o tipo que queremos usar, o PostgreSQL utiliza o B-Tree, ou seja, nas situações mais comuns. Mas podem ocorrer vezes em que o PostgreSQL escolha outro tipo, para isso, ele analisa o caso e leva em consideração se a coluna indexada está envolvida numa comparação envolvendo determinados operadores:

Tipo	Operadores
B-Tree	<, <=, =, >=, >
R-Tree	<<, &<, &>, >>, @, ~=, &&
Hash	=

No momento esta definição pode ser que fique um pouco obscura, mas por enquanto não vamos nos preocupar com isso. Em artigo próximo veremos com mais detalhes para o que são e como são utilizados os operadores e classes de operadores, vamos concentrar nossos raciocínios apenas em índices.

Criar Índice com Tipo não Padrão

Mas se desejarmos escolher o tipo de índice, devemos acrescentar ao final o comando USING, logo após especificando o tipo desejado:

CREATE INDEX *nome* ON *tabela* USING HASH (*coluna*);
CREATE INDEX *nome* ON *tabela* USING BTREE (*coluna*);
CREATE INDEX *nome* ON *tabela* USING RTREE (*coluna*);
CREATE INDEX *nome* ON *tabela* USING GIST (*coluna*);

Normalmente gera-se muita confusão no momento de escolher que índice utilizar, já que cada um tem uma finalidade. Na tabela seguinte há uma tentativa de conceituar brevemente cada um dos quatro índices:

É comum precisarmos utilizar consultas que usam na cláusula WHERE condições de pesquisas que dependem muito de duas colunas, então é necessário fazer com que o índice utilize duas colunas. Isso é totalmente possível no PostgreSQL, desde que os índices sejam do tipo B-tree ou GiST e não ultrapassem o limite de 32 colunas (o valor de 32 colunas pode ser alterado apenas durante a geração do PostgreSQL, alterando o arquivo pg_config.h em versões 7.3 ou pg_config_manual.h em versões 7.4). Quando se utiliza mais de uma coluna, o índice é organizado de acordo com a primeira coluna especificada na declaração, sendo o segundo utilizado quando a primeira coluna possuir vários valores iguais, portanto a segunda coluna surgiria como uma segunda opção de classificação.

Índice	Definição
B-Tree	São árvores de pesquisa balanceadas desenvolvidas para trabalharem em discos magnéticos ou qualquer outro dispositivo de armazenamento de acesso direto em memória secundária. O índice *B-tree* é uma implementação das árvores B de alta concorrência propostas por Lehman e Yao.
R-Tree	Também conhecido como árvores R, utiliza o algoritmo de partição quadrática de Guttman, sendo utilizada para indexar estrutura de dados multidimensionais, cuja implementação está limitada a dados com até 8Kbytes, sendo bastante limitada para dados geográficos reais. Utilizado normalmente com dados do tipo box, circle, point e outros.
Hash	O índice *hash* é uma implementação das dispersões lineares de Litwin. Na própria documentação do PostgreSQL está presente a seguinte nota: "Os testes mostram que os índices *hash* do PostgreSQL têm desempenho semelhante ou mais lento que os índices *B-tree*, e que o tamanho e o tempo de construção dos índices *hash* são muito piores. Os índices *hash* também possuem um fraco desempenho sob alta concorrência. Por estas razões, a utilização dos índices *hash* é desestimulada"** .
GiST	Generalized Index Search Trees (Árvores de Procura de Índice Generalizadas), para mais informações visite http://www.sai.msu.su/~megera/postgres/gist/doc/intro.shtml

```
CREATE INDEX coluna1_coluna2_tabela_idx ON tabela (coluna1,coluna2);
```

Também é importante salientar que quando utilizado várias colunas num índice, o otimizador de consultas pode utilizar todas as colunas especificadas ou apenas uma ou algumas, de acordo como ele decidir. Isto vai depender se as colunas são consecutivas. Por exemplo, um índice incluindo *(col_1, col_2, col_3)* pode ser utilizado em consultas envolvendo *col_1*, *col_2* e *col_3*, ou em consultas envolvendo *col_1* e *col_2*, ou em consultas envolvendo apenas *col_1*, mas não em outras combinações. (Em uma consulta envolvendo *col_1* e *col_3*, o otimizador pode decidir utilizar um índice para *col_1* apenas, tratando *col_3* como uma coluna comum não indexada).

```
CREATE TABLE clientes (id serial, nome varchar(50), ano_nasc int, valor_devido float)
CREATE INDEX ano_valor_clientes_idx ON clientes USING BTREE (ano_nasc, valor_devido);
SELECT nome FROM clientes WHERE ano_nasc > 1970 AND valor_devido < 2000;
SELECT nome FROM clientes WHERE ano_nasc > 1970 OR valor_devido < 2000;
```

No exemplo acima é criada uma tabela de importância desprezível, na qual é criado índices para as duas últimas colunas. Supondo que as tabelas já estão povoadas, são feitas duas consultas, ambas utilizando os mesmos critérios de consultas, exceto por uma utilizar o AND e outra o OR. No entanto apenas a primeira consulta utiliza o índice. Por definição, o PostgreSQL utiliza apenas o índice com mais de uma coluna quando as colunas estão unidas numa cláusula WHERE por AND, em outros casos o índice vai ser utilizado apenas na coluna que foi definida por primeiro na criação do índice. No exemplo anterior, na segunda consulta o índice *ano_valor_clientes_idx* só será usado

na procura dos clientes nascidos após 1970, pelo fato de *ano_nasc* ser a coluna principal do índice.

Os índices com mais de uma coluna devem ser evitados. Dependendo da analise há casos em que é melhor haver duas colunas do que uma. No entanto os especialistas afirmam que o uso de três ou mais colunas é praticamente inviável.*

* *Ou inevitável? (Observação de Ribamar FS).*

Neste artigo foi apenas frisado a criação de índices no PostgreSQL, o que na realidade é muito pouco para a sua compreensão. No próximo artigo continuaremos a falar do assunto, mostrando mais detalhes para que se possa tirar o melhor proveito do uso da indexação.

* Para se visualizar os índices criados na base de dados utilize o comando 'di' sem as aspas.

** Hashing: valor de identificação produzido através da execução de uma operação numérica, denominada função de hashing, em um item de dado. O valor identifica de forma exclusiva o item de dado, mas exige um espaço de armazenamento bem menor. Por isso, o computador pode localizar mais rapidamente os valores de hashing do que os itens de dado, que são mais extensos. Uma tabela de hashing associa cada valor a um item de dado exclusivo. Webster's New World Dicionário de Informática, Brian Pfaffenberger, Editora Campus, 1999.

Índices - Parte 3

Olá pessoal. Nesta série de artigos sobre índice, vimos, no primeiro artigo, conceitos básicos sobre o assunto, sem detalhar os índices no PostgreSQL. No segundo artigo tivemos a oportunidade de aprendermos como se cria índices e quais as possibilidades de índices que temos para usar. Neste terceiro artigo avançaremos mais um pouco, mostrando mais alguns detalhes importantes na sua criação. Se acaso desejar ver os dois primeiros artigos, acesse o links abaixo:

- Parte 1;
- Parte 2;

Índices únicos

Os índices não precisam ser necessariamente usados como forma de aceleração nas consultas, mas também como uma maneira de restringir que dados se repitam em uma determinada coluna. Portanto, ele funciona de forma semelhante a uma chave primária, não permitindo que, por exemplo, num campo onde se digita o cpf de um usuário sejam inseridos duas vezes o mesmo cpf. Para criar um índice que tenha esta função acrescenta-se a cláusula UNIQUE após a palavra CREATE:

```
CREATE UNIQUE INDEX nome_indice ON tabela (coluna)
```

Em casos em que são acrescentados valores nulos para a coluna, o PostgreSQL não restringe a sua inserção, podendo assim, existir vários valores nulos nesta determinada coluna da tabela. Outro detalhe importante é o de usar duas ou mais colunas com índices únicos, nestes casos apenas será impedida a inserção de dados quando todos os novos dados pertencentes ao índice estão coincidindo com uma outra tupla já existente. Num caso em que haja três colunas num índice, e se acaso for inserir um registro que repita os dados das duas primeiras colunas, mas não da terceira, o registro será inserido sem nenhum problema.

B-Tree é o único tipo de índice que aceita restrição de unicidade.

O PostgreSQL oferece esta possibilidade, mas se for usar restrição em colunas para que sejam dados únicos, é mais aconselhável utilizar as CONSTRAINTs, a utilização de índices para impor a restrição de unicidade pode ser considerada um detalhe de implementação que não deve ser acessado diretamente. Entretanto, se deve ter em mente que não há necessidade de criar índices em colunas únicas manualmente; se isto for feito, simplesmente será duplicado o índice criado automaticamente.

```
CREATE TABLE usuario
(
cpf int,
nome varchar,
CONSTRAINT constraint_exemplo UNIQUE (cpf)
) ;
```

Utilização de índices em expressões

Os índices não precisam necessariamente ficarem atrelados aos dados contidos na base de dados. Eles podem, de acordo com a necessidade, estar baseados em funções do PostgreSQL. Esta funcionalidade é útil para obter acesso rápido às tabelas baseado no resultado de cálculos.

Por exemplo, são utilizadas freqüentemente as funções lower e upper. A primeira função tem como finalidade converter os caracteres de uma string totalmente para minúsculas, e a segunda para maiúsculas. Podemos fazer comparações da seguinte forma:

```
SELECT * FROM tabela WHERE lower(coluna) = 'valor';
```

ou

```
SELECT * FROM teste1 WHERE upper(coluna) = 'VALOR';
```

Para que haja uma melhora de performance nestes tipos de consultas, podemos utilizar índices que já utilizem uma determinada função para a sua organização:

```
CREATE INDEX idx_tabela_lower_coluna ON (lower(coluna));
```

ou

```
CREATE INDEX idx_tabela_upper_coluna ON (upper(coluna));
```

Outro exemplo que podemos dar está relacionado ao fato de muitas vezes quando são inseridos dados em um determinado campo, eles podem vir com caracteres de espaço no início ou no fim da string. Para isso podem ser criados índices que automaticamente se adaptem aos registros da coluna, sem considerar os caracteres de espaço que estão sobrando. Neste caso, teremos que utilizar a função ltrim, que remove os caracteres que desejarmos no início de uma string:

```
CREATE INDEX idx_testando_ltrim_coluna
ON tabela
USING btree
(ltrim(coluna));
```

Um outro exemplo pode ser utilizado para quem habitualmente faz consultas como:

```
SELECT * FROM pessoas WHERE (nome || ' ' || sobrenome) = 'Juca Silva';
```

Então vale a pena criar um índice como:

```
CREATE INDEX idx_pessoas_nome ON pessoas ((nome || ' ' || sobrenome));
```

Se o índice for declarado como UNIQUE, este impede a criação de linhas cujos valores da coluna diferem apenas em maiúsculas e minúsculas, assim bem como linhas cujos valores da coluna são realmente idênticos. Portanto, os índices em expressões podem ser utilizados para impor restrições que não podem ser definidas como restrições simples de unicidade.

Índices parciais

Os índices parciais, como o próprio nome nos diz, não cobre uma coluna na sua totalidade, cobre apenas um subconjunto, obedecendo a uma determinada condição. Para isso, o índice conterá apenas entradas que satisfaçam a condição indicada na sua criação. O principal objetivo dos índices parciais é evitar que valores muito comuns para aquela coluna sejam indexados. A vantagem da indexação parcial, é que não será necessário atualizar o índice a cada vez que ocorrer uma atualização na tabela, apenas quando envolverem valores específicos para o índice, o que também implicará na redução do tamanho do índice e maior agilidade em consultas. Abaixo serão apresentados dois exemplos de possíveis utilizações de índices parcias, o segundo foi extraído da própria documentação do PostgreSQL:

a) Dentro de uma livraria existem uma grande quantidade de livros, sendo que existem livros escritos em português, inglês, alemão, francês e italiano. Do total destes livros, 8% são em português, 80% em inglês, 2% em alemão, 6% em francês e 4% em italiano. Mas 70% das buscas são procurando livros escritos em português, italiano, francês ou alemão e 30% em inglês.Portanto, há uma considerável quantidade de procura para livros nos idiomas que não seja o inglês.

Podemos criar um índice para o campo idioma envolvendo a coluna toda. Mas, no entanto, se as consultas são a maior parte para os idiomas em português, italiano, francês e alemão, vamos criar um índice apenas para os quatro idiomas mais usados e que possuem menos registros no banco.

Considerando que os nomes dos idiomas estão numa tabelas a parte, e inclui-se apenas o código do

idioma na tabela principal, a tabela idioma possui os seguintes valores:

Tabela idioma	
codigo	**idioma**
1	Inglês
2	Português
3	Alemão
4	Francês
5	Italiano

Usando as informações citadas acima, podemos criar um índice parcial da seguinte forma:

```
CREATE INDEX idx_livro_idioma ON livro (idioma) WHERE idioma != 1;
```

Consultas que usarem como condição ser de qualquer idioma exceto inglês , utilizará o índice parcial.

b) Suponha que as informações sobre o acesso ao servidor Web são armazenadas no banco de dados, e que a maioria dos acessos se origina na faixa de endereços de IP da própria organização, mas alguns são de fora (digamos, empregados com acesso discado). Se a procura por endereços de IP for principalmente sobre o acesso externo, provavelmente não será necessário indexar a faixa de endereços de IP correspondente à subrede da própria organização.

Para criar um índice parcial adequado ao exemplo acima, deve ser utilizado um comando como este:

```
CREATE INDEX idx_registro_acesso_ip_cliente
ON tbl_registro_acesso (ip_cliente)
WHERE NOT (ip_cliente > inet '192.168.100.0'
AND ip_cliente < inet '192.168.100.255');
```

Tipicamente, uma consulta que poderia utilizar este índice seria:

```
SELECT * FROM tbl_registro_acesso WHERE url =
'http://conteudo.imasters.com.br/1959/index.html' AND ip_cliente = inet '212.78.10.32';
```

Uma consulta que não poderia utilizar este índice seria:

```
SELECT * FROM tbl_registro_acesso WHERE ip_cliente = inet '192.168.100.23';
```

Deve-se levar em consideração que os exemplos dados anteriormente, são meramente explicativos ao que se refere os índices parciais, sem seguir ou levar em consideração conceitos de normalização de banco de dados. O PostgreSQL suporta índice parcial com predicados arbitrários, desde que somente colunas da tabela sendo indexada estejam envolvidas. Entretanto, deve-se ter em mente que a condição colocada na criação do índice deve corresponder às condições utilizadas nas consultas que supostamente vão se beneficiar do índice. O índice parcial apenas será utilizado em uma consulta se o sistema puder reconhecer que a condição WHERE da consulta implica matematicamente no predicado do índice.

O PostgreSQL não possui um provador de teoremas sofisticado que possa reconhecer predicados matematicamente equivalentes escritos de formas diferentes. O sistema pode reconhecer implicações de desigualdades simples como, por exemplo, “x < 1” implica “x < 2”; senão, a condição do predicado deve corresponder exatamente à condição WHERE da consulta, ou o índice não será reconhecido como utilizável. Normalmente, o PostgreSQL realiza escolhas racionais sobre a utilização dos índices, por exemplo, evita-os ao buscar valores com muita ocorrência, de tal forma que o segundo exemplo realmente economiza apenas o tamanho do índice, não sendo necessário para evitar a utilização do índice.

No próximo artigo veremos o que mais interessa sobre os índices, que são dicas para que os índices realmente tenham um significado positivo no seu banco de dados. Por enquanto é isso pessoal, um grande abraço a todos e até mais.

Índices - Dicas de desempenho

Olá comunidade iMasters! Depois de três artigos mostrando conceitos de índices no PostgreSQL, enfim chegamos à última parte. E nesse artigo teremos como objetivo obter idéias mais avançadas na criação de índice, tentando conduzir sempre ao melhor desempenho possível.

Para que esse objetivo seja conquistado, dependerá do responsável pela administração do banco de dados fazer una análise correta da situação que se passa com auxílio de algumas estatísticas sobre o banco. Como obter estas estatísticas ficará para ser aprofundado num próximo artigo, mas desde já comessem a se acostumar com algumas idéias interessantes. Colocarei em forma de itens uma breve seqüência de dicas:

- É importante utilizar índices em chaves estrangeiras, já que estes são muito utilizados em ***joins***. O índice será útil quando a tabela que possui uma chave estrangeira tentar acessar dados na tabela que dá suporte à esta chave. Também será útil no caso em que for excluir determinada linha de uma tabela que possui uma chave estrangeira, terá que ser feita uma leitura na tabela de onde vem os dados para a chave estrangeira para excluir os dados nesta segunda tabela no caso em que foi definido ON DELETE CASCADE, ou para não permitir a deleção na primeira tabela se caso foi determinado ON DELETE RESTRICT.

- As expressões de índice são relativamente dispendiosas de serem mantidas, uma vez que a expressão derivada deve ser computada para cada linha ao ser inserida ou sempre que for atualizada. Portanto, devem ser utilizadas somente quando as consultas que usam o índice são muito freqüentes.

- Deve-se ter muito cuidado em comparações de proporcionalidade envolvendo números relativamente pequenos. Enquanto selecionar 1.000 de cada 100.000 linhas (1% do total) pode ser um candidato para um índice, selecionar 1 de cada 100 linhas, que também corresponde a 1% do total, dificilmente será, porque as 100 linhas provavelmente cabem dentro de uma única página do disco, não havendo nenhum plano melhor que uma busca seqüencial em uma página do disco.

- Quando os índices não são usados, pode ser útil como teste forçar sua utilização. Existem parâmetros em tempo de execução que podem desativar vários tipos de planos. Por exemplo, desativar varreduras seqüenciais (enable_seqscan) e junções de laço-aninhado (nested-loop joins) (enable_nestloop), que são os planos mais básicos, forçam o sistema a utilizar um plano diferente. Se o sistema ainda assim escolher a varredura seqüencial ou a junção de laço-aninhado então existe, provavelmente, algum problema mais fundamental devido ao qual o índice não está sendo utilizado como, por exemplo, a condição da consulta não corresponder ao índice.

- Se forçar a utilização do índice não faz o índice ser usado, então existem duas possibilidades: ou o sistema está correto e a utilização do índice não é apropriada, ou a estimativa de custo dos planos de

comando não estão refletindo a realidade. Portanto, deve ser medido o tempo da consulta com e sem índices. O comando EXPLAIN ANALYZE pode ser útil neste caso.

- Quando achar necessário que determinadas informações tenham que ser únicas em uma determinada coluna, evite que a aplicação que utiliza o banco de dados faça isto. Haverá mais vantagens se o SGBD se responsabilizar pela unicidade dos dados.

- Se estiver sendo carregada uma tabela recém criada, a maneira mais rápida é criar a tabela, carregar os dados usando o COPY e, depois, criar todos os índices necessários para a tabela. Criar um índice sobre dados pré-existentes é mais rápido que atualizar de forma incremental durante a carga de cada linha. Para carregar uma tabela existente, pode-se remover o índice, carregar a tabela e, depois, recriar o índice. É claro que o desempenho do banco de dados para os outros usuários será afetado negativamente durante o tempo que o índice não existir. Deve-se pensar duas vezes antes de remover um índice único, porque a verificação de erro efetuada pela restrição de unicidade não existirá enquanto o índice não tiver sido criado novamente.

- Utilizando o comando abaixo, poderá se obter informações sobre os índices contidos no banco de dados, como por exemplo, o número total de varreduras que utilizaram um determinado índice e o número de linhas lidas com aquele índice.

SELECT * FROM pg_stat_all_indexes

- Em determinados casos, em que haja várias consultas que utilizem os comandos ORDER BY, GROUP BY e DISTINCT é aconselhável criar um índice para a coluna que está sendo utilizada nestas consultas. Isso se deve ao fato de que cada vez que ocorre isto, o SGBD dispara um SORT para a ordenação dos dados, o que pode corromper desempenho. Havendo índices para este caso, os dados já poderão estar ordenados e no final das contas economizando alguns milissegundos em processamento.

Brevemente será lançado um artigo para esclarecimento da utilização de informações estatísticas que o PostgreSQL possibilita para que se possa melhor planejar o caminho que o SGBD terá que percorrer e também para que se possa mais facilmente encontrar pontos que causem problemas no seu desempenho.

Dica:
Como já havia escrito na outra mensagem ...
Se você tiver MUITOS NULLS na tabela ou se o Indice não foi criado corretamente, ele não será usado mesmo.
Normalmente , se 30% dos valores da hash forem NULLS, o planejador não usará o indice, pq é mais rápido fazer scan.
Outra possibilidade é criar indice composto condicional (tsac AND ISNULL(tpgt)) e utilizar a funcao na comparação !
Thiago Risso na pgbr-geral.

Índices

- Tabelas pequenas não necessitam
- Como os índices têm seu custo devemos evitar muitos índices numa tabela
- Campos em cláusulas WHERE, em grandes tabelas e em consultas muito utilizadas são fortemente recomendados e geralmente melhoram muito o desempenho.
- Campos numéricos são mais indicados para índice
- Com índice vale o ditado popular: "Nem oito nem oitenta".

Teoria sobre Índices e uso no PostgreSQL:
Uma boa indicação é a série de tutoriais do Daniel Oslei, no site iMasters (http://www.imasters.com.br).

Os SGBDs utilizam índices para que consultas de recuperação, inserção, alteração e exclusão sejam feitas com melhor performance.

Criar um índice:
create index nome on tabela(campo);

Existem vários tipos de índice e o mais popular é o B-Tree. O tipo B-Tree é o único tipo que aceita restrição de unicidade.

Criar um índice de um tipo específico:
create index nome on tabela using rtree (campo);

Índice em dois campos:
create index idx_c1c2 on clientes(campo1, campo2);

Criando Índices com Funções (funcionais):
create index idx_minusculas on clientes(lower(campo));
create index idx_minusculas on clientes(upper(campo));

Índices com Expressões:
create index idx_concatena on clientes((nome || ' ' || sobrenome));

Índices Parciais:
create index idx_livro_idioma on livros(idioma) where idioma != 1;
create index idx_acesso on clientes(ip_cliente)
    where not (ip_cliente > inet '10.3.20.5'
    and ip_cliente < inet '10.5.20.6');

Informações sobre todos os índices do banco atual:
select * from pg_stat_all_indexes;

Dicas

- Sempre crie chaves primárias para as tabelas
- Sempre crie chaves estrangeiras quando indicado
- Não criar índices:
  - em tabelas com muitos nulos
  - em tabelas muito pequenas

Da FAQ do PostgreSQL:

Indexes are not used by every query. Indexes are used only if the table is larger than a minimum size, and the query selects only a small percentage of the rows in the table. This is because the random disk access caused by an index scan can be slower than a straight read through the table, or sequential scan.

To determine if an index should be used, PostgreSQL must have statistics about the table. These statistics are collected using VACUUM ANALYZE, or simply ANALYZE. Using statistics, the optimizer knows how many rows are in the table, and can better determine if indexes should be used. Statistics are also valuable in determining optimal join order and join methods. Statistics collection should be performed periodically as the contents of the table change.

Indexes are normally not used for ORDER BY or to perform joins. A sequential scan followed by an explicit sort is usually faster than an index scan of a large table. However, LIMIT combined with ORDER BY often will use an index because only a small portion of the table is returned.

If you believe the optimizer is incorrect in choosing a sequential scan, use `SET enable_seqscan TO 'off'` and run query again to see if an index scan is indeed faster.

When using wild-card operators such as LIKE or ~, indexes can only be used in certain circumstances:

- The beginning of the search string must be anchored to the start of the string, i.e.
  - LIKE patterns must not start with *%*.
  - ~ (regular expression) patterns must start with ^.
- The search string can not start with a character class, e.g. [a-e].
- Case-insensitive searches such as ILIKE and ~* do not utilize indexes. Instead, use expression indexes, which are described in section 4.8.
- The default *C* locale must be used during *initdb* because it is not possible to know the next-greatest character in a non-C locale. You can create a special `text_pattern_ops` index for such cases that work only for LIKE indexing. It is also possible to use full text indexing for word searches.

How do I perform regular expression searches and case-insensitive regular expression searches? How do I use an index for case-insensitive searches?

The ~ operator does regular expression matching, and ~* does case-insensitive regular expression matching. The case-insensitive variant of LIKE is called ILIKE.

Case-insensitive equality comparisons are normally expressed as:

```
SELECT *
FROM tab
WHERE lower(col) = 'abc';
```

This will not use an standard index. However, if you create an expression index, it will be used:

```
CREATE INDEX tabindex ON tab (lower(col));
```

If the above index is created as UNIQUE, though the column can store upper and lowercase characters, it can not have identical values that differ only in case. To force a particular case to be stored in the column, use a CHECK constraint or a trigger.

Links dos originais no iMasters:
http://imasters.uol.com.br/artigo/1897/postgresql/indices/
http://imasters.uol.com.br/artigo/1922/postgresql/indices_-_parte_02/
http://imasters.uol.com.br/artigo/1959/postgresql/indices_-_parte_3/
http://imasters.uol.com.br/artigo/1977/postgresql/indices_-_dicas_de_desempenho/

Arquitetura Interna de Banco de Dados - Índices

Autor: Prof. Vandor Roberto Vilardi Rissoli

Nos arquivos sequencias o compromisso de manter os registros fisicamente ordenados pelo valor da chave de ordenação, acarreta uma série de cuidados.

A operação de inserção, por exemplo, pode conduzir à necessidade do uso de áreas de extensão, além de periódicas reorganizações do BD.

Com o objetivo de conseguir um acesso eficiente aos registros, pode ser acrescentada uma nova estrutura para uso do BD, sendo ela definida como ÍNDICE.

O uso de índice é conveniente aos processos que precisam de um desempenho mais ágil, mesmo para os registros que não se encontrem ordenados fisicamente.

Exemplo:
Para uma consulta que deseje encontrar todas as contas somente da agência de Brasília têm-se que:

- refere-se a uma porção (ou fração) de todos os registros de contas;
- é ineficiente para o sistema ler todos os registros para localizar somente os que são desta agência;
- seria eficiente um acesso direto a esses registros;

Assim torna-se interessante acrescentar uma nova estrutura que permita essa forma de acesso.

O atributo, ou o conjunto de atributos, usado para procurar um registro em um arquivo é chamado de chave de procura ou chave de acesso.

Esta definição de chave difere das definições de chaves estudadas até agora (candidata, primária, ...). Por meio das chaves de procura será possível acrescentar em um arquivo várias chaves de acesso, sendo elas disponibilizadas de acordo com a necessidade existente na situação.

Geralmente, é interessante ter mais que um índice para um arquivo.

Exemplo:

• Imagine o catálogo de livros em uma biblioteca. Caso você deseje encontrar um livro de um autor específico, esta pesquisa será realizada sobre o catálogo de autores.

• O resultado desta pesquisa fornecerá um cartão que identificará onde encontrar o livro.

• Para ajudar na pesquisa sobre o catálogo, a biblioteca mantém os cartões em ordem alfabética.

• Com isso não será necessário pesquisar todos os cartões do catálogo para encontrar o cartão desejado.

• Em uma biblioteca são diversos os catálogos de procura (autor, título, assunto, entre outros).

Essa nova estrutura adicional consiste na definição de índices para os arquivos de dados, definindo os arquivos indexados.

nos arquivos indexados podem existir tantos índices quantas forem as chaves de acesso aos registros;

um índice irá consistir de uma entrada para cada registro do arquivo, sendo que as entradas encontram-se ordenadas pelo valor da chave de aceso;

cada índice é formado pelo valor de um atributo chave do registro e pela sua localização física no interior do arquivo;

a estrutura com a tabela de índices é também armazenada e mantida em disco.

Ainda assim, existirão grandes arquivos de índices que não seriam manipulados tão eficientemente (como no exemplo anterior "biblioteca").

Por isso, algumas técnicas mais sofisticadas de índices são adotadas.

Dois tipos básicos de índices:

Ordenados: baseiam-se na ordenação dos valores

Hash: baseiam-se na distribuição uniforme dos valores determinados por uma função (função de hash)

Não existe uma técnica melhor, sendo cada técnica mais adequada para aplicações específicas.

Cada técnica deve ser avaliada sobre alguns fatores:

- tipos de acesso: encontrar registros com um atributo determinado ou dentro de uma faixa de valores
- tempo de acesso: tempo gasto para encontrar um item de dados ou um conjunto de itens
- tempo de inserção: tempo gasto para incluir um novo item de dados, incluindo o tempo de localização do local correto e a atualização da estrutura de índice
- tempo de exclusão: tempo gasto para excluir um item de dados, sendo incluído o tempo de localização do registro, além do tempo de atualização do índice
- sobrecarga de espaço: espaço adicional ocupado pelo índice, compensando este sacrifício pela melhoria no desempenho

ÍNDICES ORDENADOS

Para conseguir acesso aleatório rápido sobre os registro de um arquivo, pode-se usar uma estrutura de índice.

Cada índice esta associado a uma chave de procura (armazena a chave de procura de forma ordenada e associa a ela os registros que possuem aquela chave);

Os registros em um arquivo indexado podem ser armazenados (eles próprios) em alguma ordem;

Um arquivo pode ter diversos índices, com diferentes chaves de procura, estando ele ordenado sequencialmente pelo seu índice primário;

Normalmente, os índices primários em um arquivo são as chaves primárias, embora isso nem sempre ocorra

Índice Primário

Este índice consiste em um arquivo ordenado cujos registros são de tamanho fixo, contendo dois campos:

1º- mesmo tipo do campo chave no arquivo de dados

2º- ponteiro para o bloco do disco

O índice primário é um índice seletivo, pois possui entradas para um subconjunto de registros (bloco), ao invés de possuir uma entrada para cada registro do arquivo de dados.

Como o índice primário mantém ordenado os registros no arquivo, os processos de inserção e remoção são um grande "problema" para este tipo de estrutura.

ÍNDICE ORDENADO

Há dois tipos de índices ordenados (denso e esparso):

Índice Denso

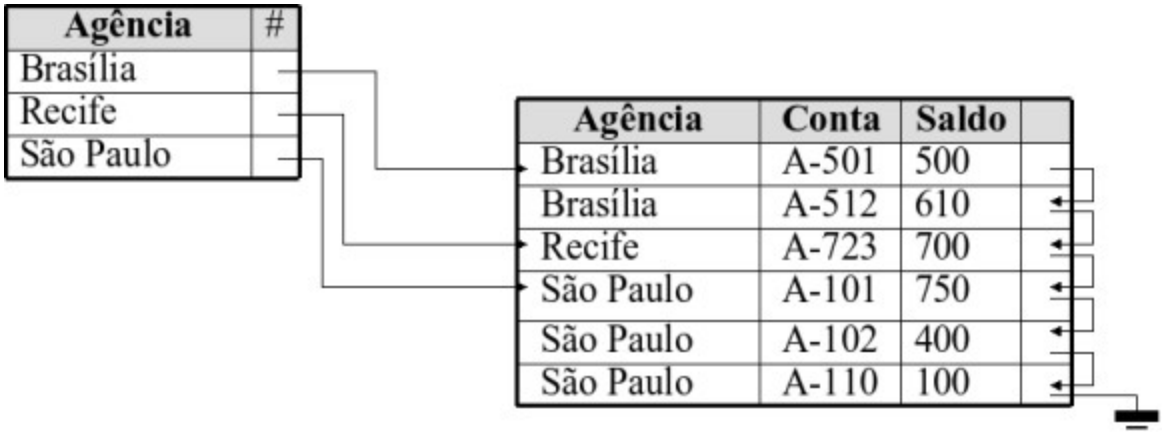
Um registro de índice aparece para cada valor distinto da chave de procura no arquivo;

O registro contém o valor da chave de procura e um ponteiro para o primeiro registro de dados com esse valor;

Alguns autores usam esta expressão para identificar quando um registro de índice aparece para cada registro no arquivo de dados.

Exemplo:

Índice denso para as agências bancárias.



Índice Esparso

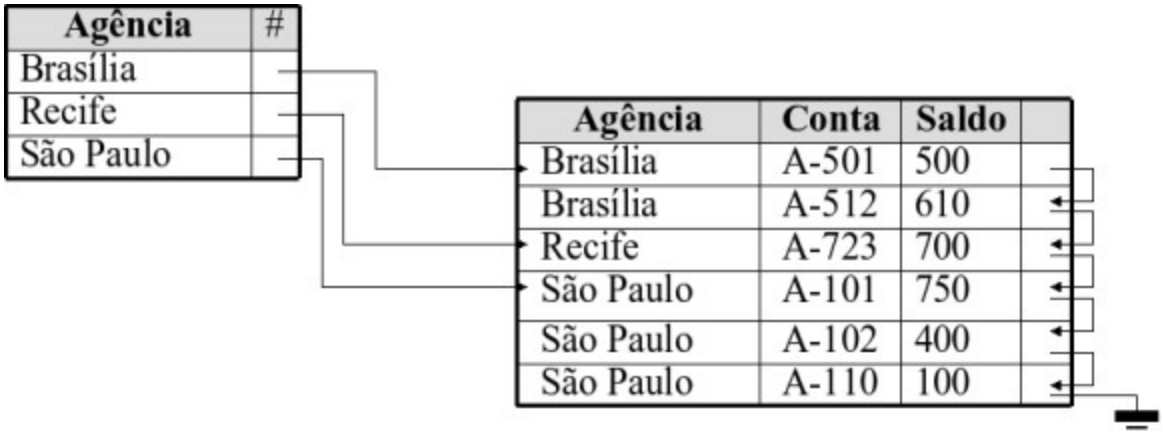
Um registro de índice é criado apenas para alguns dos valores do arquivo de dados;

Assim como os índices densos, o registro contém o valor da chave de procura e um ponteiro para o primeiro registro de dados com esse valor;

Localiza-se a entrada do índice com o maior valor da chave de procura que seja menor ou igual ao valor da chave de procura que se deseja. Inicia-se no registro apontado e segue-se pelos ponteiros até localizar o registro procurado.

Exemplo:

Índice esparso para as agências bancárias.



Uma nomenclatura também utilizada é a de índice de cluster, onde tem-se os registros de um arquivo, fisicamente ordenados por um campo não chave, podendo serem distintos ou não.

Índice Secundário

Consiste de um arquivo ordenado que não usa o mesmo campo de ordenação como índice.

São os índices cujas chaves de procura especificam uma ordem diferente da ordem sequencial do arquivo, podendo seus valores serem distintos para todos os registros ou não.

Os índices secundários melhoram o desempenho das consultas que usam chaves diferentes da chave de procura do índice primário, mas impõem uma sobrecarga significativa na atualização do BD.

O projetista do BD decide quais índices secundários são desejáveis baseado na estimativa da frequencia de consultas e atualizações.

ÍNDICES DE NÍVEIS MÚLTIPLOS

Arquivo de índice muito grande para ser eficiente;

Registros de índices são menores que os registros de dados;

Índices grandes são armazenados como arquivos seqüenciais em disco.

 Uma busca em um índice grande também é muito onerosa.

Uma possível solução na agilização de grandes Índices

Criação de índice ESPARSO como índice primário;

Realização de busca binária sobre o índice mais externo (encontrar maior valor da chave de procura que seja menor ou igual ao valor desejado);

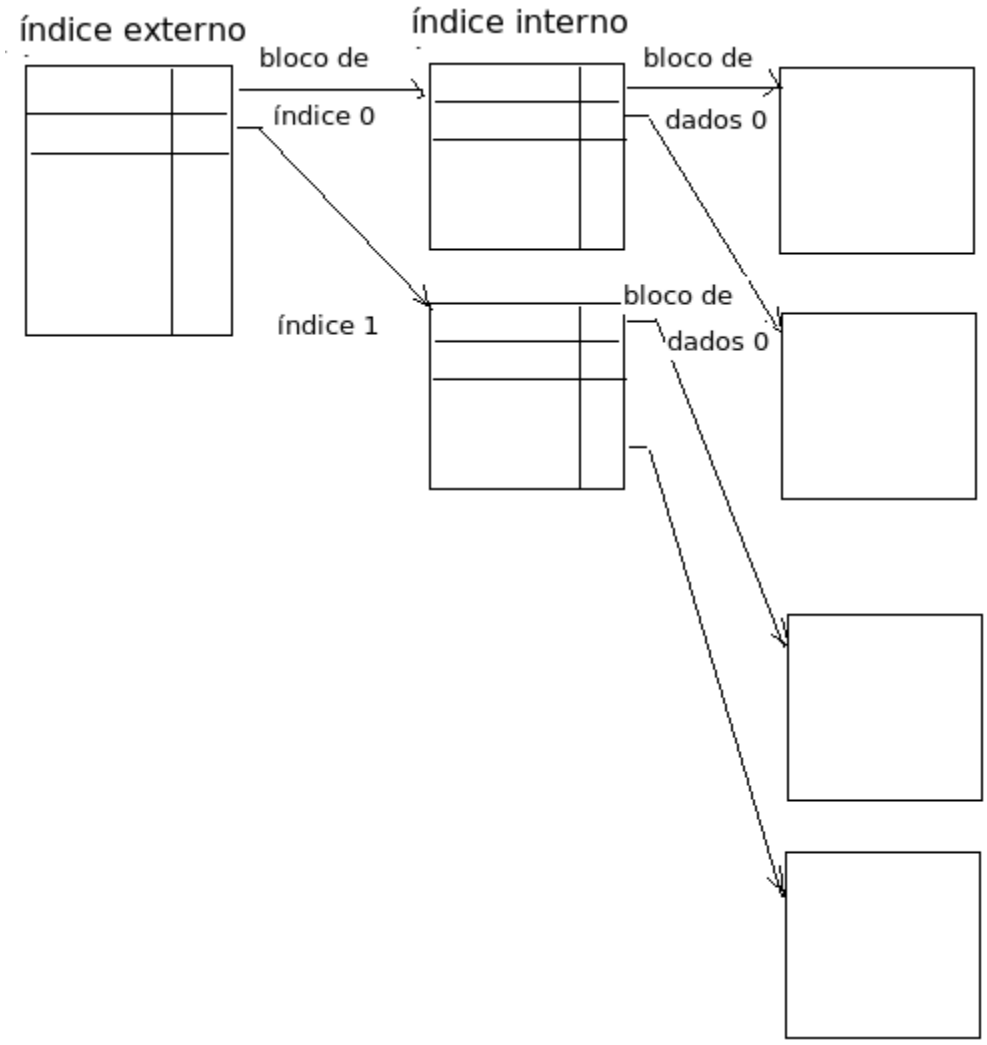
O ponteiro indica o bloco do índice interno que será “varrido” até se encontrar o maior valor da chave de procura que seja menor ou igual ao valor desejado;

O ponteiro neste registro indica o bloco do arquivo que contém o registro procurado.

Observe a representação a seguir:

 Utiliza-se dois níveis de indexação;

- caso o nível mais externo ainda seja muito grande;
- não cabendo total-mente na memória principal;
- pode ser criado um novo nível;
- criar quantos níveis forem necessários.


índice externo
índice interno
bloco de
índice 0
bloco de
dados 0
índice 1
bloco de
dados 0

BUSCA ou PROCURA usando Níveis Múltiplos

Requer um número significativamente menor de operações de E/S que a busca binária;

Cada nível do índice corresponde a uma unidade de armazenamento físico (trilha, cilindro, ou disco);

 Os níveis de índices múltiplos estão relacionados as estruturas de árvores, como as árvores binárias usadas na indexação de memória.

ATUALIZAÇÃO DE ÍNDICE

Independente do tipo de índice utilizado, ele deve ser atualizado sempre que um registro for inserido ou removido do arquivo de dados.

REMOÇÃO

Localizar o registro;
Remover o registro;
(se houver)
Atualizar o índice;
(denso ou esparso)

INSERÇÃO

Localizar o registro;
(usa chave de procura)
-Incluir os dados (registro)
Atualizar o índice
(denso ou esparso)

ORGANIZAÇÃO DE ARQUIVOS COM HASHING

Para uma localização de dados eficiente, em arquivos sequenciais, é necessário o acesso a uma estrutura de índice ou o uso da busca binária, que acarreta em mais operações de E/S.

Outra alternativa seria a aplicação de técnicas de hashing, onde os arquivos seriam organizados por meio de uma função de hash.

Com o uso destas técnicas, evitaria-se o acesso a uma estrutura de índice;

O hashing também proporciona um meio para construção de índices;

A organização de arquivos hashing oferece:

Acesso direto ao endereço do bloco de disco que contém o registro desejado;

Aplica-se uma função sobre o valor da chave de procura do registro para conseguir-se a identificação do bloco de disco correto;

Utiliza-se do conceito de bucket (balde) para representar uma unidade de armazenamento de um ou

mais registros;

Normalmente equivalente a um bloco de disco, porém ele pode denotar um valor maior ou menor que um bloco de disco.

FUNÇOES HASH

Uma função Hash ideal distribui as chaves armazenadas uniformemente por todos os buckets, de forma que todos os buckets tenham o mesmo número de registros.

No momento do projeto, não se sabe precisamente quais os valores da chave de procura que serão armazenados no arquivo.

Deseja-se que a função de hash atribua os valores da chave de procura aos buckets atentando a uma distribuição:

•UNIFORME

•ALEATÓRIA (o valor de hash não será correlacionado a nenhuma ordem visível externamente – alfabética por exemplo – parecendo ser aleatória)

Observe a escolha de uma função hash sobre o arquivo conta, usando a chave de procura nome da agência.

Decidiu-se a existência de 27 buckets (por estado);
Função simples, mas não distribui uniformemente os dados (SP tem mais conta que MA), além de não ser aleatória;

Neste outro exemplo, aplica-se as características típicas das funções de hash.

Efetuam cálculos sobre a representação binária interna dos caracteres da chave de procura;
Uma função simples calcularia a soma das representações binárias dos caracteres de uma chave de procura, retornando o módulo da soma pelo número de buckets;

CONCLUSÃO SOBRE AS HASH

O emprego destas funções requerem um projeto cuidadoso.

Função Hash “RUIM”: pode resultar em procuras que consumam um tempo proporcional a quantidade de chaves no arquivo;

Função Hash “BOA”:oferece um tempo de procura médio (constante pequena), independente da quantidade de chaves de procura no arquivo;

OVERFLOW DE BUCKET

Até o momento supôs-se que os buckets sempre tem espaço para inserir um novo registro, porém isso pode não acontecer. Quando o bucket não possuir espaço suficiente, diz-se que ocorreu um overflow de bucket.

Este overflow pode acontecer por várias razões, sendo algumas delas apresentadas a seguir:

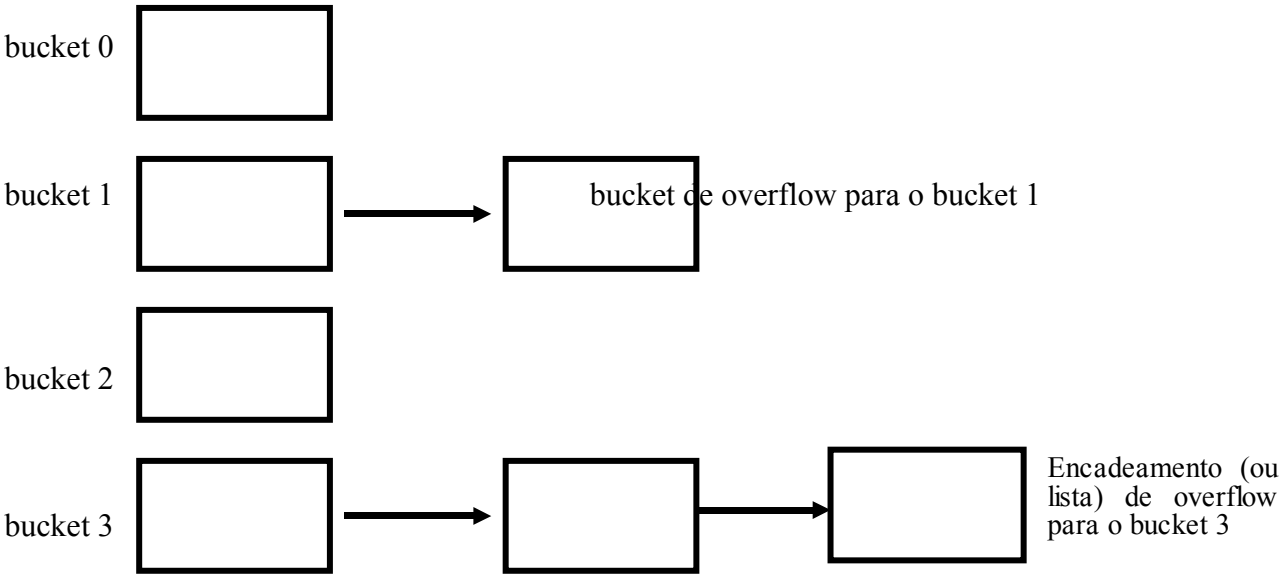
Buckets insuficientes: conhecer o total de registro quando a função de hash for escolhida;

Desequilíbrio (skew): pode acontecer por duas razões:

- registros múltiplos com a mesma chave de procura;
- distribuição não uniforme realizada pela função de hash escolhida.

Baseado em alguns cálculos é possível reduzir a probabilidade de overflow de buckets, processo este conhecido como fator de camuflagem (ou fator de fudge).

Algum espaço será desperdiçado no bucket, cerca de 20% ficará vazio, mas o overflow será reduzido. Ainda sim, o overflow de bucket poderá acontecer, por exemplo:



Cuidados com o algoritmo de procura para o overflow:

A função de hash é usada sobre a chave de procura para identificar qual o bucket do registro desejado;

Todos os registros do bucket devem ser analisados, quando a igualdade com a chave de procura acontecer, inclusive nos bucket de overflow;

Existem outras variações na aplicação das técnicas de hashing (hashing aberto, linear probing, ...), porém a técnica descrita anteriormente é preferida para banco de dados (aspectos referentes aos problemas com a remoção).

Desvantagens desta Técnica

A função de hash deve ser escolhida até o momento no qual o sistema será implementado;

Como a função de hash mapeia os valores da chave de procura para um endereço fixo de bucket, ela não poderá ser trocada facilmente, caso o arquivo que esteja sendo indexado aumente muito ou diminua;

ATIVIDADE EM GRUPO

Temas (livro do Korth capítulo 11):

1- Arquivos de Índice Árvore B+ (pág.347 – 11.3)
2- Arquivos de Índice Árvore B (pág.356 – 11.4)
3- Índices Hash (pág.362 – 11.5,2)
4- Arquivos Grid (pág.374 – 11.9.1 e 11.9.2)

Atividade (painel integrado):

• desenvolver o estudo dos temas pelo grupo
• discutir e anotar os conteúdos mais relevantes
• trocar os membros de cada grupo que explicaram aos novos companheiros de grupo o material estudado

INDEXAÇÃO ORDENADA X HASHING

Cada esquema representa vantagens e desvantagens para determinadas situações (inclusive para os arquivos heap – arquivos que não são ordenados de nenhuma maneira em particular);

Tipo de consulta também é um fator importante:
Igualdade (... where atributo = valor)
em hash o tempo médio de procura é uma constante independente do tamanho do BD;

Faixa de valores (... where atributo < valor)
na ordenada localiza-se o valor e retorna-se todos os valores seguintes do bucket até atingir o valor limite, enquanto que o hash é aleatório e terá que localizar cada valor;

Material para Consulta e Apoio ao Conteúdo

ELMASRI, R. e NAVATHE, S. B., Fundamentals of Database Systems - livro
Capítulo 6

SILBERSCHATZ, A., KORTH, H. F., Sistemas de Banco de Dados - livro
Capítulo 11

Melhorando o Desempenho de Consultas com Adição de Índice

1) Melhorar o desempenho de consultas, adicionando índices aos campos da cláusula where (em consultas
que retornam muitos registros), de tabelas com muitos registros, como exemplo:

Observe que a tabela de ceps não está normalizada, quase todos os campos deveriam pertencer a outras
tabelas, como é o caso de tipo, logradouro, bairro, municipio e uf. Como também vários campos deveriam conter
índices para agilizar as consultas.

Exemplifiquemos adicionando um índice em uf, mas antes de adicionar mediremos o desempenho e após também.

2) Criar uma estrutura semelhante à do arquivo .csv que queremos importar:

```
create table cep_full
(
	cep char(8),
	tipo char(72),
	logradouro char(70),
	bairro char(72),
	municipio char(60),
	uf char(2)
);
```

3) Importar para a tabela criada todo o conteúdo do CSV (Comma Separated Values):

Linux:
```
\copy cep_full from /home/ribafs/cep_brasil_unique.csv
```

Windows:
```
\copy cep_full from c:/teste/cep_brasil_unique.csv
```

4) Fazer uma cópia da tabela cep_full:
```
create table cep_full_index as select * from cep_full;
```

5) Adicionar chave primária à tabela criada:
```
alter table cep_full_index add constraint cep_pk primary key(cep);
```

Vamos planejar uma consulta, mas antes atualizemos as estatísticas internas do planejador:
```
analyze cep_full_index;
```

6) Testar um plano de consulta pela UF:

```
explain select count(uf) from cep_full_index where uf='CE';
```

Analize os resultados. Uma observação importante é o fato de não usar índice, mas uma busca sequencial.
Veja também os valores dos custos, que são relativamente altos.

Observe que o comando explain não executa a consulta, apenas mostra o plano interno de execução da mesma,
por isso seu resultado é instantâneo por maior que seja a tabela.

Vamos então criar um índice para o campo uf para ver o resultado:
```
create index idx_uf on cep_full_index(uf);
```

Agora repetir a execução do plano de consulta anterior para ver como se comporta com índice:
```
explain select count(uf) from cep_full_index where uf='CE';
```

Veja que os custos da função count cairam pela metade enquanto que os costos da varredura, que agora usa
índice, cairam de 35.457 para 173, o que representa menos de 0.5%.

Agora faremos um teste aparentemente mais realista:

```
create table t1 as select * from cep_full;
alter table t1 add constraint cep_pk primary key(cep);
analyze t1;
\timing
select count(uf) from t1 where uf='CE';

create index idx_uf on t1(uf);
select count(uf) from t1 where uf='CE';
```

Veja que gastou em torno de 10% de quando não havia índice.

Então, sempre que se deparar com uma consulta em tabelas com grande quantidade de registro não exite em usar índices nos campos do where. Caso ainda tenha alguma dúvida já sabe o que fazer, elabore um plano de execução de duas consultas (uma com e outra sem índice no campo)e veja o que
o PostgreSQL fará nos dois casos.

Teoria sobre Índices e uso no PostgreSQL:
Uma boa indicação é a série de tutoriais do Daniel Oslei, no site iMasters (http://www.imasters.com.br).

Garantindo desempenho com o operador LIKE
De Rodrigo HJort na SQL Magazine 52

Consultas em campos string que contenham acentos não melhoram com a criação de índices simples.
Para isso devemos criar índices especiais:

A criação de índice comun em campos nem leva a uma busca indexada pelo planejador.

```
create index nome_idx on clientes(nome varchar_pattern_ops);
```

Com o índice acima realmente podemos melhorar o desempenho de consultas por campos acentuados em LATIN1.

Veja que ainda existe um outro porém, para consultas por esse campo (nome) usando LIKE, apenas será
melhorado o desempenho se usarmos uma consulta do tipo:

```
select * from clientes
        where nome LIKE 'JOSÉ CARLOS %';
```

Com o caractere coringa % no fonal da string de busca.
Caso o caractere esteja no início já não resolverá.

Tem mais: caso desejemos usar alguma função como upper ou lower na busca, estas funções

deverão fazer
parte do índice para que tenhamos um desempenho melhorado, ou seja, para que o planejador utilize o
índice.

Quando o caractere coringa está no ínicio da string de busca devemos encontrar uma forma para que o
planejador use o índice e uma delas é criar um índice reverso.

Opção - criar uma função em plperl:

```
create language plperl;

create or replace function reverse(varchar) returns varchar as $$
   return reverse $_[0];
$$ language plperl immutable strict;

select reverse('Joao Brito Cunha');

create index idx_nome2 on clientes (reverse(nome));
create index idx_nome3 on clientes (reverse(nome) varchar_pattern_ops);
alanyze clientes;
```

Nos testes o planejador irá utilizar o índice idx_nome3, devido ser LATIN1.

Obs.: Um índice reverso irá ler os nomes do final para o começo, ao contrário do índice normal. "Joao" será lido como "oaoJ".

Caso utilizemos o caractere coringa tanto no início quanto no final da string de busca, então os métodos
atuais não ajudarão em termos de desempenho.