**Otimização de consultas SQL**

**Otimizar o servidor de banco de dados** é muito importante, mas [**melhorar o desempenho de consultas individuais**](http://www.devmedia.com.br/sql-server-profiler-otimizando-consultas-no-sql-server/30665) pode ser ainda mais satisfatório. Existem várias formas de otimizar o banco e as consultas. Muitas vezes os bancos não são bem projetados e/ou não estão normalizados. Problemas de informações redundantes ou relacionamentos mal definidos provenientes de bases não normalizadas podem afetar de forma significante o desempenho das consultas existentes.

**Processamento do SQL**

Para ser eficaz em [**instruções SQL**](http://www.devmedia.com.br/principais-instrucoes-em-sql/5748), o profissional de banco de dados deve ter uma compreensão profunda sobre como o SQL é transformado a partir do código fonte original em uma forma executável.Em um nível alto, o processamento SQL é dividido em várias etapas:

1. Análise do código fonte para localizar erros de sintaxe;

2. Uso do otimizador SQL para obter um plano de execução;

3. O script é executado com base no plano elaborado;

4. Busca do conjunto de resultados do banco de dados e resposta à consulta chamada.

Geralmente, a maior parte das linguagens de programação executam suas instruções de cima para baixo. Porém, [**o SQL Server**](http://www.devmedia.com.br/guias/sql-server/27) executa em uma ordem única, que é conhecida como fase lógica de processamento de consulta. Estas fases originam uma série de tabelas virtuais com cada uma destas alimentando a fase seguinte (tabelas virtuais ocultas). Estas fases e suas ordens são dadas como observado na**Listagem 1**.

**Listagem 1.**Etapas do processamento lógico do SQL

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | (8) SELECT (9) DISTINCT (11) <TOP quantidade> <lista de campos de retorno>   (1) FROM <tabela>   (3) <tipo de junção> JOIN <tabela> ON (2) <condições da junção>   (4) WHERE <condições where >   (5) GROUP BY <lista de agrupamento>   (6) WITH {CUBE | ROLLUP}   (7) HAVING <condições having>  (10) ORDER BY <lista de campos> |

A seguir são descritos os itens visualizados na listagem e o que ocorre em cada fase do processamento:

1. FROM: um produto cartesiano (cross join) é realizado entre as duas primeiras tabelas na cláusulafrome como resultado a tabela virtual VT1 (tabela virtualizada) é gerada;

2. ON: é aplicado em VT1. Somente as linhas para as quais a <condições da ‘junção> é verdadeira serão inseridas em VT2:

2.1. OUTER (junção): se for especificado umouter join(ao contrário de umcross joinou um inner join), as linhas da tabela preservada ou a tabela que não foi encontrada uma correspondência são adicionados às linhas de VT2 como linhas exteriores, gerando o VT3. Se mais de duas tabelas aparecem na cláusulafrom, as etapas 1 a 3 são aplicadas várias vezes entre o resultado da última associação e a próxima tabela na cláusulafrom, até todas as tabelas serem processadas.

3. WHERE: é aplicado a VT3. Apenas as linhas para as quais a <condições where> é verdadeira são inseridas a VT4;

4. GROUP BY: as linhas da VT4 são organizadas em grupos com base na lista de colunas especificada na cláusulagroup by, então VT5 é gerado;

5. CUBE | ROLLUP: supergrupos (grupos de grupos) são adicionados às linhas da VT5 gerando VT6;

6. HAVING: é aplicado a VT6. Apenas os grupos para os quais a  
<condições having>é verdadeira são inseridos à VT7;

7. SELECT: é processada gerando VT8;

8. DISTINCT: linhas duplicadas são removidas da VT8 gerando a VT9;

9. ORDER BY: as linhas da VT9 são classificadas de acordo com a lista de coluna especificada na cláusulaorder by. Um cursor é gerado (VC10);

10. TOP: o número especificado ou porcentagem de linhas são selecionadas a partir da  
VC10. A tabela VT11 é gerada e retorna ao executor.

As etapas básicas descritas para o processamento de uma instrução select se aplicam a outros comandos SQL como insert, update e delete.O processo de identificação dessas linhas é semelhante ao procedimento utilizado para identificar as linhas de origem que colaboram para o conjunto de resultados de uma instrução select.As instruções update e insert podem conter instruções select, incorporadas, que fornecem os valores de dados a serem atualizados ou inseridos.

### Índices

Se suas consultas realizam buscas em tabelas grandes, uma boa recomendação é [**o uso de índices**](http://www.devmedia.com.br/entendendo-e-usando-indices-parte-1/6567). Um índice é uma estrutura em disco associada a uma tabela ou exibição que agiliza a recuperação de linhas. Usá-los é uma tomada de decisão adequada para resolver a maior parte dos problemas de consultas com tempo demorado. Todas as chaves primárias precisam de índices para realizar as junções com outras tabelas de forma mais eficaz. Isso faz com que todas as tabelas precisem de uma chave primária.

Índices são normalmente criados em colunas que são acessadas ​​com maior frequência de modo que a informação possae ser recuperada mais rapidamente. Os índices podem ser criados em uma única coluna ou em um grupo delas. Quando um índice é criado, ele primeiro classifica os dados e, em seguida, atribui um Rowid (chave única e sequencial) para cada linha.

### Índices clusterizados e não clusterizados

A diferença básica entre índices de cluster e sem cluster é que os registros de um índice de cluster são classificados e armazenados sequencialmente com base em sua chave. O banco cria automaticamente índices quando uma restrição primary key ou unique em uma tabela é criada:

· Índice clusterizados: um índice clusterizado determina a sequência de armazenamento dos registros em uma tabela. Eles são usados ​​para campos em que são realizadas buscas frequentes ou que são acessados ​​de forma ordenada. Uma tabela pode ter somente um índice clusterizado (pela necessidade de ordenação do campo indexado);

· Índice não-clusterizado: os dados são armazenados em local diferente do índice e os ponteiros indicam o armazenamento de objetos indexados na tabela. Este tipo de índice é utilizado quando se realizam buscas em campos onde os dados sejam únicos. Os campos dos tipos text, ntext, image não podem receber esse tipo de índice. Para resolver esse problema, pode-se aplicar um índice em uma view. De uma forma geral, os índices do tipo não-clusterizados devem ser utilizados quando os campos são: frequentemente usados nos critérios de pesquisa; usados para se juntar a outras tabelas; usados como campos de chave estrangeira ou na cláusula order by.

### Views indexadas

As views indexadas podem ser usadas pelo banco de dados de duas maneiras diferentes. Primeiro, a view pode ser chamada a partir de uma consulta (que é usada convencionalmente). Basicamente, o comando é executado, utilizando um índice agrupado para mostrar os resultados da view quase que imediatamente. Em segundo lugar, em qualquer consulta que é executada, o gerenciador do banco de dados automaticamente avalia se existem índices relacionados à view. Se assim for, o otimizador de consulta usa o índice existente, mesmo que não tenha sido especificado na consulta, para assim aumentar a velocidade de execução.

O primeiro índice criado em uma view deve ser um índice clusterizado exclusivo.Depois que este tipo for criado, você poderá criar índices não-clusterizados.Criar um índice clusterizado exclusivo em uma view melhora o desempenho da consulta porque a view é armazenada no banco de dados da mesma forma que uma tabela com um índice clusterizado é armazenada.

Para exemplificar o uso desse tipo de recurso, na**Listagem 2** é criada uma tabela de log com um milhão de registros. Esta tabela contém o identificador do visitante, a data da visita e do valor da operação que informa se o usuário fez uma compra durante a visita.

**Listagem 2.**Criação da estrutura do exemplo de utilização da view indexada

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | CREATE TABLE dbo.LOG(    ID\_LOG int NOT NULL IDENTITY(1,1)    COOKIE int NOT NULL,    DATA\_VISITA date NOT NULL,    VALOR money NOT NULL   )    CREATE CLUSTERED INDEX IDX\_LOG\_DATA\_VISITA    ON dbo.LOG (DATA\_VISITA ASC)    WITH (PAD\_INDEX = OFF, STATISTICS\_NORECOMPUTE = OFF, SORT\_IN\_TEMPDB = OFF,  DROP\_EXISTING = OFF, ONLINE = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON) ON [PRIMARY]    --insere 1 milhão de registros  DECLARE @i int = 0, @total int = 1000000  WHILE (@i < @total)  BEGIN    INSERT INTO dbo.LOG (COOKIE,DATA\_VISITA,VALOR)    SELECT RAND()\*@total/100 --COOKIE        , DATEADD(day,CONVERT(INT,(RAND()\*500)),"20130101") –-DATA VISITA        , CASE WHEN RAND()<0.01 THEN RAND()\*10.0 ELSE 0.0 END --VALOR      SET @i+=1  END |

Na **Listagem 3**executa-se um exemplo de consulta para informar o número de visitas, o valor total de compras e a receita por visita de cada cookie.

**Listagem 3.**Consulta de exemplo

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | SELECT COOKIE    , <span class="lf-badge">count</span>(\*) AS FREQUENCIA    , <span class="lf-badge">sum</span>(VALOR) AS VALOR\_TOTAL    , AVG(VALOR) AS VALOR\_MEDIA  FROM DBO.LOG  GROUP BY COOKIE |

Em média, esta consulta levou 577 milissegundos nos testes executados, mas pode-se acelerá-la usando um índice não clusterizado como observado no comando a seguir:

|  |  |
| --- | --- |
| 1  2  3 | <p align="left">    CREATE NONCLUSTERED INDEX [IDX\_LOG\_VW\_LOG] ON [LOG] ([COOKIE])    INCLUDE ( [VALOR]) ON [PRIMARY] |

Ao executar novamente a consulta apresentada, ela é executada agora 212 milissegundos, uma melhoria de 63,25% no desempenho. Entretanto, pode-se aumentar esse ganho de desempenho usando uma view indexada em vez de criar o índice anterior, como demonstrado na **Listagem 4**.

**Listagem 4**. Criação da view indexada.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | CREATE VIEW [VW\_LOG] WITH SCHEMABINDING AS    SELECT COOKIE      , <span class="lf-badge">sum</span>(VALOR) AS VALOR      , <span class="lf-badge">count</span>\_BIG(\*) AS FREQUENCIA    FROM DBO.LOG    GROUP BY COOKIE    CREATE UNIQUE CLUSTERED INDEX IDX\_VW\_LOG ON [VW\_LOG] (COOKIE); |

Executando novamente a consulta da **Listagem 3**, ela roda agora em 56 milissegundos em média, um ganho de desempenho de 90,29%.

Mesmo que a média agregada (avg) não esteja definida na view, o otimizador de consulta é capaz de obter o resultado utilizando os valores do count e do sum. Caso a view tenha uma quantidade grande de dados, pode-se também criar índices não clusterizados para aumentar a velocidade de acesso às informações.

Views indexadas também são uma ótima forma de melhorar o**desempenho de inner joins**. Quando duas ou mais tabelas se relacionam em uma view indexada, o otimizador de consulta pode escolher recuperar os dados diretamente da view em vez de executar uma custosa operação de junção.

### Outras práticas com melhor performance de execução

Existem outras particularidades que podem ser adotadas para que as consultas tenham uma melhor performance.

### Uso de Union

Um comando union equivale a fazermos a junção de dois conjuntos eliminando, em seguida, os elementos duplicados (o que poderia ser feitos através de um comando distinct). Se sabemos que existem registros duplicados e isso representa um problema para a aplicação, então devemos utilizar o union para eliminá-los. Por outro lado, se não haverá linhas duplicadas ou se não é um problema tê-las, utiliza-se o union all em vez de union. A vantagem do union all é que ele não realiza o distinct, evitando o desperdício de recursos do servidor SQL.

Para exemplificar, veja a **Listagem 5**. Imagine que se quer realizar uma consulta para mesclar dois conjuntos de dados.

**Listagem 5.** Exemplo do comando Union.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | (1)  SELECT nome\_columa1, nome\_columa2     FROM tabela1     WHERE nome\_columa1 = value     UNION     SELECT nome\_columa1, nome\_columa2     FROM tabela1     WHERE nome\_columa2 = value    (2)  SELECT DISTINCT nome\_columa1, nome\_columa2     FROM tabela1     WHERE nome\_columa1 = value OR nome\_columa2 = value |

A consulta marcada com o número (2) é executada de forma mais rápida do que a (1). E isso pode ser melhorado (sabendo que a união desses dois conjuntos de dados contém elementos duplicados) removendo o comando distinct.

### Relação entre tabelas

É bastante comum realizar uma comparação e relação entre tabelas. Na **Listagem 6** são exemplificadas três formas de executar essa operação.

**Listagem 6.** Exemplo de relação entre tabelas.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | (1)  SELECT a.nome\_columa1     FROM tabela1 a     WHERE NOT EXISTS (SELECT b.nome\_columa2 FROM tabela2 b WHERE b.nome\_columa2   = a.nome\_columa1)    (2)  SELECT a.nome\_columa1     FROM tabela1 a       LEFT JOIN tabela2 b ON b.nome\_columa2   = a.nome\_columa1     WHERE b.nome\_columa2 IS NULL    (3)  SELECT nome\_columa1     FROM tabela1     WHERE nome\_columa1 NOT IN (SELECT nome\_columa2 FROM tabela2) |

Em cada uma das consultas apresentadas o resultado é o mesmo. Porém, qual delas tem a melhor performance? Assumindo que todo o resto é igual, a versão que tem o melhor desempenho é a primeira (1) e a última (3) é a pior. O comando not exists (ou exists) é o mais eficiente.

### Uso do comando Group By

A cláusula group by pode ser usada com ou sem uma função agregada (max, sum, count, avg, ...). Para obtermos um melhor desempenho, não devemos utilizá-la sem uma função agregada. Observe a **Listagem 7**.

**Listagem 7.** Utilização do comando Group by.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | (1)  SELECT nome\_columa1, nome\_columa2     FROM tabela1     WHERE nome\_columa1 > value     GROUP BY nome\_columa1, nome\_columa2    (2)  SELECT DISTINCT nome\_columa1, nome\_columa2     FROM tabela1     WHERE nome\_columa1 > value |

Ambas as consultas retornam os mesmos resultados, porém a segunda obtém um melhor desempenho. Para melhorar a performance ao utilizarmos a cláusula group by, deve-se considerar as seguintes recomendações:

· O número de linhas de retorno a partir da consulta deve ser o menor possível;

· Manter o número de agrupamentos o mais limitado possível;

· Não agrupar colunas redundantes;

· Se existe um join na mesma instrução select que tem um group by, tente reescrever uma consulta utilizando uma subconsulta em vez de usar o join. Se for possível fazer isso, o desempenho será melhor. Se for necessário usar um join, utilize as colunas do group by com a mesma coluna da tabela em que a função está sendo usada;

· Considere adicionar um order by para a(s) mesma(s) coluna(s) existente(s) no group by. Isso pode fazer com que ele tenha um melhor desempenho.

A**otimização de consultas** é um trabalho extremamente importante para a gestão e manutenção de uma base de dados. A finalidade de se otimizar consultas e gerenciar as estruturas e índices de um banco é livrar os usuários de suas complicações e das exigências indispensáveis para se obter consultas eficientes.