Guia de Melhores Práticas de Desenvolvimento Performance em Banco de Dados Oracle e SQL Server

Preparado por: José Carlos, Paulo Santoro

Data: 27/08/19

1 Sumário

O objetivo deste documento é buscar, através de Melhores Práticas de Desenvolvimento e Performance em Banco de Dados Oracle e SQL Server, obter ganhos de desempenho, além de sugerir melhores formas de Implementar, desenhar e monitorar os sistemas SEFAZ.

Este guia pode ser usado por qualquer time responsável pelo desenvolvimento de sistemas e modelagem de dados.

É recomendado que este sirva de base para o desenvolvimento de sistemas com maior qualidade, evitando o uso de certas práticas que levam a um mau desempenho do banco de dados.

Este manual se aplica a todas as atuais versões do Oracle e SQL Server.

Objetivo:

Definir um conjunto de boas práticas para o desenvolvimento em banco de dados.

Público Alvo:

Desenvolvedores de aplicativos da secretaria da fazenda.

Escopo:

Banco de dados Microsoft SQL Server e Oracle.

1 Codificação SQL

1.1 Utilizar o Visual Studio Database Project e o SQL Developer (na solução da aplicação) para construção e alteração de código SQL;

1.2. Utilizar procedures ao invés de querys explícitas na aplicação;

1.3. Não utilizar stored procedures genéricas, evitando o excesso de operadores "OR" para combinação de filtros;

1.4. Ao construir consultas defina as colunas selecionadas, evitando a sintaxe " SELECT * ";

1.5. Considerar o uso de melhores práticas na criação dos índices;

1.6. Adotar práticas de tratamento de erros em stored procedures, como o uso de TRY/CATCH por exemplo;

1.7. Os scripts de banco de dados devem ser gerados automaticamente através de ferramenta (Data Tools/Data Modeler) de comparação de schemas;

2 Processamento de arquivos (ETL)

2.1. Devem ser utilizados pacotes SSIS/ODI para o processamento de arquivos;

2.2. Dentro do pacote SSIS, deve-se fazer uso de Bulkinsert (fast load) para carga de dados;

2.3. Os pacotes SSIS devem fazer uso de fontes de dados adequados, evitando utilizar script task/component para execução de operações nativas. Exemplo: Não usar script para conectar no SQL como fonte de dados, usar fonte de dados Oledb;

2.4. Deve-se utilizar o CONTROL+M para agendamento e execução de pacotes SSIS/ODI;

2.5. Deve-se adotar uma solução de log de erros na execução dos pacotes SSIS/ODI;

3 Salvar imagens e documentos

3.1. Utilizar FILESTREAM (Microsoft) e CLOB/BLOB (Oracle) para salvar imagens e documentos;

3.2. Havendo pesquisa em documentos, utilizar full text (Microsoft) e Context Index (Oracle).

4 Processamento de mensagens

4.1. SQL Service Broker é a ferramenta de mensageria padrão no ambiente da SEFAZ.

5 Modelagem de dados

5.1. Devem-se seguir os padrões de uso de tipo de dados;

5.2. Devem-se seguir os padrões de nomenclatura;

5.3. Modelos de Dados na "terceira forma normal".

6 Escolha de tipos de dados

6.1. Utilizar o tipo de dado de menor tamanho para cada coluna de acordo com a necessidade de negócio;

6.2. Escolher tipo de dado que evite a conversão de tipo (implícita ou explícita), isto pode ter um custo no desempenho das consultas;

6.3. Evitar valores nulos em campos de chaves estrangeiras, limitando o número de "outer joins" que precisarão ser escritos;

6.4. Preferir (Microsoft) o tipo VARCHAR ao invés do tipo TEXT (o tipo TEXT tem custo adicional porque é armazenado em páginas de texto/imagem ao invés de páginas de dados);

6.5. Evitar (Microsoft) o tipo SQL_VARIANT, pois implica em custo adicional por armazenar metadados (pode impactar o desempenho na conversão de tipos);

6.6. Considerar que o tipo de dado Unicode (NCHAR ou NVARCHAR) ocupa o dobro do espaço do tipo ASCII (CHAR ou VARCHAR);

6.7. Não utilizar os tipos de dados VARCHAR e NVARCHAR para colunas com tamanho menor que 4 caracteres;

6.7.1 Evitar o Uso de VARCHAR / NVARCHAR (MAX), somente em casos específicos, pois ele se comporta como um tipo de dado TEXT ou VARBINARY, e nesse caso não podem ser compactados (COMPRESS) no ambiente produtivo;

6.8. Utilizar o tipo de dado XML somente nos seguintes casos:

6.8.1. Quando dados semiestruturados são necessários;

6.8.2. Quando parte de uma entidade não pode ser estruturada com técnicas padrão de modelagem de dados;

6.8.3. Quando parte de uma entidade já existente é um XML e precisa ser armazenado nativamente;

6.8.4. Quando parte do conteúdo do XML necessita ser pesquisado. Considerar duplicar campos do XML em colunas na tabela. Assim poderemos criar um índice sobre a coluna.

7 Campos Identificadores

7.1. No Microsoft SQL server o desenvolvedor deverá gerar na aplicação (na camada cliente ou na camada de negócios) quando existir a necessidade de inserção de registro numa tabela que tenha um campo UNIQUEIDENTIFIER.

Evitar a geração pelo SQL Server como propriedade default do campo.

No caso de necessidade de geração em stored procedures, deve ser utilizada a função newid() para a criação do valor;

7.2. Casos mais simples (incrementos sequenciais) utilizar sequences tanto no Microsoft quanto no Oracle, por recomendação da própria MS, estamos entrando em desuso do "Identity";

8 Campos nulos

8.1. O desenvolvedor deverá evitar o uso de conteúdo nulo nas colunas. Só deverão ser aceitos quando for um requisito de negócio;

8.2. Como prática recomendada, colunas que serão utilizadas como critério de busca não devem aceitar nulo. Se uma coluna de entrada de pesquisa permite nulo, a interface de usuário deve fornecer um meio para indicar isto.

9 Índices

9.1. O desenvolvedor deve analisar e criar os índices conforme a previsão de uso pela aplicação;

9.2. O desenvolvedor deve criar índices somente quando necessários. Apesar de melhorar a performance nas consultas, ele prejudica a performance dos comandos DML (insert, update e delete). Portanto, devem ser usados, mas com critério;

9.3. Os índices ocupam espaço no disco. É mais um fator a ser considerado na hora de criá-los;

9.4. Seguem alguns critérios para a criação de índices:

9.4.1. Criar índices baseado no uso;

9.4.2. Manter índices clustered (Microsoft) tão pequenos quanto possível;

9.4.3. Criar índices para as chaves estrangeiras, salvo raras exceções;

9.4.4. Criar índices com alto nível de seletividade;

9.4.5. Criar índices compostos com a coluna mais restritiva em primeiro lugar;

9.4.6. Considerar a criação de índices para colunas utilizadas em cláusulas "WHERE", "ORDER BY", "GROUP BY" e "DISTINCT".

9.4.7. Os índices deverão ser criados nos locais (filegroups e tablespaces) adequados. Em tabelas particionadas deverão respeitar preferencialmente as regras de particionamento;

9.4.8. Para campos XML:

9.4.8.1. Considerar o alto consumo de espaço de um índice XML (até quatro vezes o tamanho da coluna). Portanto, utilizar somente quando for realmente necessário;

9.4.8.2. Quando muitos campos do XML precisam ser pesquisados;

9.4.8.3. Existem múltiplas ocorrências num único XML;

9.4.8.4. Necessidade de verificar a existência de uma entidade.

10 Chaves Estrangeiras

10.1 As chaves estrangeiras devem ser modeladas para não aceitar valores nulos;

10.2 Devem estar habilitadas com o objetivo de garantir a integridade referencial entre as tabelas;

10.3 Auxiliam o otimizador de consultas criar o melhor plano de execução.

11 Funções ou User Defined Function (UDF)

11.1. Somente deve ser criada pelo desenvolvedor quando não houver uma função equivalente nativa no banco de dados.

12 User Defined Types/Datatypes (UDT)

12.1 O desenvolvedor deve evitar o uso de UDTs.

13 Esquemas

13.1 O desenvolvedor deve criar todos os objetos de banco de dados em um esquema apropriado. O esquema irá auxiliar na divisão dos objetos em agrupamentos lógicos.

13.2. Utilizar as seguintes recomendações:

13.2.1. Referir a um objeto utilizando seu FQN (full qualified name) ou utilize o nome do esquema seguido do nome do objeto, separado por um ponto (.);

13.2.2. Simplificar a implementação de esquemas através do uso de sinônimos para abstrair o proprietário do esquema dos objetos;

13.2.3. Gerenciar permissões de usuários no nível de esquema. Isto ajuda a proteger os objetos de banco de dados;

13.2.4. Utilizar esquemas para combinar entidades relacionadas num único banco de dados físico.

14 Sinônimos

14.1. O desenvolvedor deve utilizar sinônimos para tornar transparente o acesso aos objetos do banco de dados.

15 Sequences

15.1. Criar sequences com o objetivo de popular colunas auto incrementadas.

16 Views

16.1. O desenvolvedor deve utilizar views para auxiliar no desenvolvimento de banco de dados, principalmente para construir consultas que possuam múltiplas junções entre diferentes tabelas.

16.2 Utilizar as seguintes recomendações:

16.2.1. Evitar o uso de "Hint";

16.2.2. Sempre analisar o plano de execução da view.

17 Views Indexadas ou Views Materializadas

17.1. O desenvolvedor deve utilizar essas views para aumentar o desempenho das consultas. No entanto, elas somente devem ser criadas em casos especiais e implantadas com muito critério;

17.2. O desenvolvedor deve consultar a documentação do fornecedor para utilizá-las de forma adequada.

18 Stored Procedures

18.1. O desenvolvedor deve utilizar stored procedures para acesso/atualização dos dados. Ela não deve conter regras de negócio, que devem estar na camada de negócio da aplicação.

18.2 Utilizar as seguintes recomendações:

18.2.1. Quando a stored procedure retornar somente um registro, utilizar parâmetros de saída ao invés de um conjunto de linhas;

18.2.2. No Microsoft SQL Server sempre utilizar a cláusula SET NOCOUNT ON e não utilizar comandos PRINT em produção. Também evitar o uso do prefixo "sp_" no nome da Stored Procedure;

18.2.3. Retornar somente as colunas e os registros que serão utilizados pela aplicação.

19 SQL Dinâmico

19.1. O desenvolvedor deve evitar o uso de SQL dinâmico;

19.2. No caso de não haver alternativa, para o Microsoft SQL Server é preferível utilizar o comando sp_executesql ao invés do comando EXEC ('string'). Isto para evitar risco de SQL injection e permitir o reuso de plano de execução.

20 Cursores

20.1. O desenvolvedor deve evitar o uso de cursores. Eles são utilizados para processamento registro a registro de uma tabela. Na maioria dos casos, podemos obter o mesmo resultado por meio de comandos SQL;

20.2. Caso o uso de cursor seja imprescindível, é necessário garantir o fechamento e desalocação do cursor. No Oracle, alternativamente pode ser utilizado o BULK COLLECT e o FORALL.

21 Transações

21.1 O desenvolvedor deve manter as transações mais curtas quanto possível. Isto reduz o número de locks e o risco de deadlock.

22 Transaction Isolation Level

22.1. O desenvolvedor deve evitar alterar o nível de isolamento padrão do database;

22.2. No caso do Microsoft SQL server é comum o desenvolvedor utilizar SET TRANSACTION ISOLATION LEVEL ou o NOLOCK com o objetivo de alterar o nível de isolamento para READ UNCOMMITED.

Esse procedimento não é recomendado, apesar de minimizar problemas de bloqueios, pois a transação poderá recuperar dados inconsistentes. Optar pelo nível de isolamento READ COMMITTED, que é o padrão.

Ele proporciona um bom equilíbrio entre consistência e concorrência.

O uso do nível de isolamento SNAPSHOT deve ser considerado quando um grande número de leituras precisa ser realizado em tabelas que possam ter alterações.

As modificações sobre os dados realizadas por outras transações após o início da transação corrente não são visíveis aos comandos executados durante a transação.

As transações SNAPSHOT não requerem locks para leitura de dados, não bloqueando outras transações.

23 Particionamento de tabelas

23.1. O desenvolvedor deve considerar particionar tabelas nos seguintes casos:

23.1.1. Otimizar a performance de consultas em tabelas muito grandes (acima de 10 milhões de registros) ou em tabelas com grande previsão de crescimento;

23.1.2. O ganho de performance ocorre quando utilizamos a chave do particionamento como um dos argumentos de pesquisa. Dessa maneira, somente serão acessados os registros de uma determinada partição;

23.1.3. Facilitar o gerenciamento de tabelas muito grandes, principalmente nos casos de exclusão de um grande conjunto de registros. A manipulação de dados de apenas uma partição Evita a interrupção do acesso a tabela;

23.1.4. Reduzir o tempo de indisponibilidade em situações de desastre;

23.1.5. Os índices, quando possível, devem conter a chave de particionamento, assim, também poderão ser particionados;

23.1.6. Alocar, preferencialmente, cada partição em um Filegroup (Microsoft) ou Tablespace (Oracle);

23.1.7. O desenvolvedor deve consultar a documentação do fornecedor para implantar as regras de particionamento de forma adequada.

24 Planos de Acesso

24.1. O desenvolvedor deve se preocupar com os planos de acesso de todas as pesquisas, especialmente as mais utilizadas pelo sistema;

24.2. Os índices, conforme já abordado, são excelentes ferramentas para melhorar a performance; Contudo, deverão ser utilizados com parcimônia;

24.3. Verificar se as estatísticas dos objetos envolvidos foram atualizadas antes de analisar os planos de acesso;

24.4. O desenvolvedor deve consultar a documentação do fornecedor para analisar os planos de acesso de forma adequada.

Agora por último vão 25 Dicas para o dia a dia dos desenvolvedores:

1. Utilização de EXISTS

Caso precise retornar em uma consulta registros de uma tabela que satisfaçam uma determinada condição segundo referências de uma segunda tabela, ao invés de utilizar uma subconsulta na cláusula WHERE para um operador IN prefira a utilização de **EXISTS**:

```
SELECT * FROM MINHA_TABELA M  WHERE EXISTS
(
SELECT * FROM TABELA_LOG L WHERE L.ID_MINHA_TABELA = M.ID AND TO_CHAR(L.DATA, 'YYYY') = '2017'
)
```

Na maior parte dos cenários, esta forma tem um desempenho muito superior, em diversos bancos dados, do que a utilização da cláusula IN com subconsultas:

2. Utilize flags de tipo booleano ou inteiro

Caso tenha necessidade de possuir colunas em uma tabela cujo valor se refira à valores verdadeiros e falsos, como por exemplo, especificar se um registro está ativo ou não na sua tabela, opte por criar as colunas com o tipo booleano nativo da base de dados específica que você está utilizando, ao invés de utilizar informações 'S' ou 'N' em um campo do tipo texto. Em tabelas com muitos registros, o filtro por estes campos torna-se extremamente lento.

O Oracle infelizmente não possui um tipo booleano como o SQL Server e outros bancos relacionais. Então a saída neste caso poderia ser utilizar um campo do tipo NUMERIC (1) para armazenar 1 ou 0. Pelo menos é mais performático do que 'S' ou 'N'.

3. Conversões com UPPER, TO_CHAR e etc. em cláusulas WHERE

Evite fazer conversões de tipo e formato em colunas na cláusula WHERE para realizar filtro de dados. Esta operação faz com que o banco de dados naturalmente ignore a utilização dos índices automáticos criados para estas colunas, que tornariam a consulta bem mais rápida. Estude sempre a possibilidade de já armazenar os dados no formato correto ou que tenha uma predominância na forma de visualização na aplicação.

4. Não utilize HAVING para filtrar dados

Caso necessite filtrar dados em um agrupamento de informações, prefira sempre realizar esta operação na cláusula WHERE ao invés do HAVING, por questões de performance, a não ser que seja necessário realizar algum filtro utilizando realmente as operações de agregação:

```
--NÃO RECOMENDÁVEL
SELECT NOME, TIPO FROM MINHA_TABELA A  GROUP BY NOME, TIPO HAVING TIPO = 2
--RECOMENDÁVEL

SELECT NOME, TIPO FROM MINHA_TABELA A  WHERE TIPO=2 GROUP BY NOME, TIPO
```

5. Evite atribuir valores em variáveis com DUAL

Quem utiliza o banco de dados Oracle no seu dia a dia está muito habituado a utilizar o DUAL em muitas situações, mas eu recomendaria não utilizá-lo em atribuições de variáveis em procedures, pois isto representa perda de performance em sistemas críticos:

```
declare x number;
   begin
                -- NÃO RECOMENDÁVEL
                select 1 into x from dual;
                -- RECOMENDÁVEL
                x := 1;

   end;
```

6. Tome cuidado com MaxValue das SEQUENCES

No banco de dados Oracle, ao criar um objeto do tipo SEQUENCE, coloque na propriedade MAXVALUE um valor extremamente alto para que sua aplicação não pare de funcionar por ter atingido o valor máximo. A atribuição de um valor alto não acarreta em alocação de espaço desnecessário, pois é apenas uma informação de configuração do objeto do tipo SEQUENCE.

7. Influência da orientação a objetos na modelagem de base de dados

Caso o programador não tenha muita experiência em modelagem de banco de dados, ficando restrito ao código da aplicação, é comum haver uma tendência de ser influenciado a "pensar" em orientação a objetos ao modelar uma base de dados. Embora a utilização de ORM tenha facilitado muito a vida dos desenvolvedores, é sempre importante frisar que os bancos de dados relacionais não são orientados a objetos, mesmo podendo haver semelhanças entre tabela x objetos, colunas x propriedades e por aí vai.

A base de dados deve ser projetada e modelada segundo boas práticas de banco de dados e não boas práticas de orientação a objetos.

8. Utilize procedures e views

Quando não utilizamos procedures e views, toda vez que executamos uma instrução SQL é necessário que o database analise se a sintaxe do comando está correta, se os objetos referenciados realmente existem, dentre outras checagens necessárias para a execução do comando.

Quando o código a ser executado encontra-se em uma procedure ou view, o banco de dados não precisa fazer estas verificações e validações, pois as mesmas já foram feitas no momento de criar as procedures e views. Portanto, com o banco de dados poupando este trabalho, logicamente a performance das execuções de SQL enviados pela aplicação aumenta consideravelmente em sistemas críticos.

9. Tipos de dados são extremamente importantes. Se preocupe sempre com os tipos das colunas das tabelas do sistema. Verifique se os mesmos correspondem fielmente ao tipo de informação que será armazenada. Por exemplo, se a coluna irá armazenar uma data, crie um campo do tipo DATE. Se vai armazenar um número inteiro, crie uma coluna do tipo INTEGER e etc. Isto parece óbvio, mas é muito comum encontrarmos este tipo de situação. Assim, garantiremos que não serão inseridos dados incompatíveis ou inconsistentes com o tipo de dado da coluna da tabela. Isso melhora significativamente a performance por não precisar fazer conversões de tipo de dado.

10. Traga no SELECT somente as colunas necessárias

Esta é a dica mais básica de todas: evite utilizar SELECT * FROM. É muito comum fazermos consultas com JOINS em diversas tabelas. Especificar no SELECT apenas as colunas que serão utilizadas. Outro benefício importante é facilitar a leitura do SQL em manutenções.

11. Utilização de Cache

Caso se utilize na aplicação consultas de informações que não são frequentemente atualizadas, considere a possibilidade de colocar estes dados em cache para poupar o banco de dados deste trabalho. Mas, esta opção deve ser sempre analisada considerando o cenário de cada projeto e seus pré-requisitos.

12. Tipos das variáveis e parâmetros

Procure utilizar nas variáveis e parâmetros de procedures e de funções exatamente os mesmos tipos das colunas da tabela para evitar conversões desnecessárias.

13. Só utilize ORDER BY e DISTINCT se for realmente necessário

Muitas vezes, para determinadas funcionalidades do sistema, ordenação é algo dispensável, portanto, deve ser evitada. As vezes queremos, por exemplo, apenas listar o conteúdo de uma coluna na tela e resolvemos por conta própria, sem especificações da área de negócio, fazer ordenação por data, registro mais recente e etc. Muitas vezes isso não agregará valor ao usuário final. É estranho algum retorno de consulta sem ordenação? Pode até parecer, mas deve ser utilizado conscientemente por questões de performance, assim como o DISTINCT.

14. Configuração de linguagem, idioma e cultura

O ideal é que o banco de dados utilizado pela aplicação esteja configurado em relação ao idioma/cultura compatível com as regras de negócio ou contexto do sistema para evitar ter que realizar conversões explícitas nas consultas de banco de dados, gerando prejuízo à performance. Tais configurações estão relacionadas a aspectos de globalização. No caso do Oracle diz respeito ao **National Language Support.**

15. Utilização de "VALUES multi-row"

Caso necessite realizar múltiplas inserções em uma mesma tabela de forma sequencial, é preferível utilizar a sintaxe conforme abaixo, pois gerará ganho de performance em sistemas críticos e economizará algumas linhas de código:

```
--RECOMENDÁVEL
INSERT INTO MINHA_TABELA (CAMPO1, CAMPO2)VALUES
(1, 'Teste1'),
(2, 'Teste2'),
(3, 'Teste3')

--NÃO RECOMENDÁVEL
INSERT INTO MINHA_TABELA (CAMPO1, CAMPO2) VALUES (1,'Teste1');
INSERT INTO MINHA_TABELA (CAMPO1, CAMPO2) VALUES (2,'Teste2');
INSERT INTO MINHA_TABELA (CAMPO1, CAMPO2) VALUES (3,'Teste3');
```

16. Monitoramento de querys

Mesmo após o sistema estar em produção ou ainda em homologação, você como parte integrante do time de desenvolvimento, procure participar ativamente com o time de infraestrutura e banco de dados, para auxiliar no monitoramento e análises das operações do banco de dados aos projetos que você participa, a fim de identificar pontos de melhoria ou antecipar possíveis problemas. É fundamental estimular contato entre os times. Isto está ficando cada vez mais comum, pois é uma premissa básica para DevOps.

17. Utilização de índices em colunas muito acessadas

Caso identificar a necessidade da criação de algum índice que vise melhorar a performance das consultas de uma aplicação, procure fazer as seguintes perguntas para determinar o critério de criação, exatamente nesta ordem:

Qual coluna é acessada ou requisitada com mais frequência, sendo chave-primária ou não?

Será que não é conveniente a modificação ou remodelagem da estrutura para fins de performance, considerando a criticidade desta minha consulta?

18. Cuidado ao utilizar índices em colunas que são atualizadas com muita frequência

A utilização de índices nem sempre é uma boa alternativa em determinados cenários. Um deles é quando o índice é criado em colunas que sofrem atualizações com muita frequência. Mesmo que o desenvolvedor vise melhorar a performance nas consultas, ele pode piorar a performance nos comandos DML (insert, delete e update). Portanto, criação de índices é algo que deve ser sempre analisado com muito cuidado.

19. Índices em colunas muito presentes em WHERE, JOIN, ORDER BY e TOP

Uma boa dica para verificar se seria conveniente a criação de um índice em determinada coluna é verificar a frequência de utilização delas em cláusulas WHERE, JOIN, ORDER BY e TOP. Esta é sempre uma boa pista de índices que poderiam ser criados.

Mas, como informado no tópico anterior, a criação de índices sempre deve ser analisada e aplicada com muito cuidado.

20. Não deixe as chaves estrangeiras para depois

Esta dica é quase que tão óbvia e trivial quanto a não utilização de SELECT * FROM. Mas, é muito comum vermos sistemas utilizando tabelas sem os devidos relacionamentos no banco de dados. Então, nunca deixe pra depois a criação das chaves primárias e estrangeiras. Crie-as juntamente com a criação da tabela.

21. Utilização otimizada das tabelas de log e históricos

É comum ter tabelas de logs com milhões de registros que são muito pouco ou quase nunca acessadas. Estude a possibilidade de armazenar boa parte do histórico em outras tabelas (arquivo morto), permanecendo nas tabelas de histórico e de log somente os registros mais recentes. Mas, ressalto que cada caso deve ser analisado de forma específica, pois muitas vezes os requisitos de negócio demandam que todo o histórico e logs estejam facilmente acessíveis nas tabelas originais e não em um "arquivo morto".

22. Insira comentários

Ao criar uma tabela ou coluna no banco de dados não economize nos comentários a respeito de seu significado, fazendo isto na própria base de dados, principalmente se o sistema for legado. Os comentários na base de dados são mais importantes do que os comentários na aplicação. Eles facilitam significativamente a interpretação e a manutenção pelos desenvolvedores que precisam lidar com a base de dados.

23. Tabelas sem chave-primária

Sim, infelizmente isto existe aos montes por aqui. Se a sua tabela não possui chave primária, é recomendável que seja feita revisão na sua modelagem, pois em teoria uma tabela não deveria ficar "isolada" num modelo relacional.

24. Dedicação de tempo à modelagem

Como dito anteriormente, muitas vezes o banco de dados é a "alma" do sistema. Vale a pena investir tempo no correto planejamento e modelagem da base de dados, refletindo na estrutura a ser criada de cada tabela, coluna, relacionamentos e muitos outros aspectos.

Investir nesta etapa vale muito a pena. Certamente evitará muitas dores de cabeça no futuro.

25. UPDATE sem WHERE

E para finalizar: não execute um UPDATE sem WHERE, principalmente se você estiver em **PRODUÇÃO!**

Conclusão

Melhores práticas no desenvolvimento de aplicações e modelagem de banco de dados.
Embora muitas vezes os desenvolvedores prefiram dedicar seus esforços na construção de código-fonte, eles devem ter ciência da importância que um bom projeto de banco de dados tem para o sucesso das suas aplicações. Quase todos os sistemas envolvem acesso e persistência de dados, portanto, as estruturas de armazenamento representam a "alma" do sistema em relação ao valor agregado do negócio. As questões de segurança, de performance, de políticas organizacionais, dentre outros fatores, tornam esta "camada" extremamente relevante e merecedora de uma atenção especial por parte dos desenvolvedores.

Contudo, a equipe de Banco de Dados de Desenvolvimento da SEFAZ, Equipe CDS-DBA, está sempre à disposição para ajudá-los e orientá-los naquilo que precisarem.
Qualquer dúvida nas criações de índices, os particionamentos das tabelas e verificação dos planos de execuções, podem nos procurar, estaremos sempre prontos para ajudar.