Refatoração de Bancos de Dados Relacionais: Uma abordagem prática

TRABALHO DE CONCLUSÃO DE CURSO

Fabrízio de Royes Mello
Pós-Graduação em Tecnologias Aplicadas a Sistemas de Informação com Métodos Ágeis
Centro Universitário Ritter Dos Reis - UNIRITTER
fabriziomello@gmail.com

Guilherme Silva de Lacerda
Professor Orientador
guilherme_lacerda@uniritter.edu.br

Abstract

Refatoração é um processo de mudança na estrutura do código fonte de um software sem alterar sua semântica, e inicialmente foi utilizado em código orientado a objeto mas não ficando limitado a apenas este contexto. Em bancos de dados relacionais também são utilizadas técnicas de refatoração para melhorar sua estrutura interna sem modificar seu comportamento externo, entretanto devem ser tratados de forma diferente porque um banco de dados além de estrutura existem os dados, sem contar que inúmeras aplicações podem acessar o mesmo banco de dados dificultando ainda mais a aplicação da refatoração. Este trabalho apresenta uma abordagem prática do processo de refatoração de um banco de dados relacional usando como exemplo um modelo simplificado.
***Palavras-chave**: Refatoração, Banco de Dados, Orientação Objetos, Semântica.*

1 Introdução

Uma estrutura de um banco de dados, diferentemente da estrutura de um software, tende a deteriorar naturalmente com o passar do tempo. Dentre várias causas de deterioração podemos citar o crescimento progressivo do volume de dados devido ao aumento natural de usuários que o utilizam e também ao seu próprio tempo de uso, tornando um modelo de dados que no início era eficiente para solução proposta em um modelo ineficiente e defasado.

Essa deterioração natural aliada a mudanças em requisitos de negócio exigem modificações e refatorações tanto no software que os implementa quanto em seus bancos de dados. Entretanto a refatoração de um banco de dados é mais complexa que a de um software devido aos seguintes motivos: (i) além de manter comportamento também é necessário manter as informações (dados) e (ii) acoplamento com diversas origens (outras aplicações, *frameworks*, integrações, etc) [1].

Devido a essas dificuldades a evolução de uma estrutura de banco de dados torna-se um desafio, ocorrendo assim um fenômeno conhecido como *Bad Smells* (mal cheiros), da mesma forma que ocorre com o código de um software. Em software um *code smell* (*bad smell*) é uma categoria comum de problema no código fonte que indica a necessidade de refatoração [2], e o mesmo ocorre com bancos de dados, onde são chamados *database smells* [1].

O presente trabalho consiste em uma abordagem prática de uso de técnicas de *database refactoring* que podem ser utilizados tanto para evolução do *schema* de um banco de dados quanto para eliminação de *smells*.

2 Revisão de Literatura

2.1 Database Refactoring

Refatoração de código (*Code Refactoring*) é uma disciplina/processo que consiste em melhorar a estrutura interna de um software sem modificar seu comportamento externo [2], e uma Refatoração de Banco de Dados (*Database Refactoring*) parte do mesmo princípio, porém além de manter o comportamento externo também deve manter a semântica da informação que ele mantém/armazena, e por esse motivo é considerada mais difícil [1].

Pode-se então considerar que um *Database Refactoring* é uma mudança disciplinada na estrutura de uma base de dados que não altera sua semântica, porém melhora seu projeto e minimiza a introdução de dados inconsistentes [3].

Conforme última definição minimizar a introdução de dados inconsistentes é um dos grandes objetivos de se realizar uma refatoração na estrutura de um banco de dados, ou seja, melhorar o desing atual para melhorar a consistência dos dados e também a qualidade dos novos dados que serão adicionados.

E esta tarefa não é simples, pois existe um fator preponderante no que diz respeito a dificuldade de execução deste tipo de refatoração que é o **acoplamento**.

2.1.1 Acoplamento

É a medida de dependência entre dois elementos. Quanto mais acoplados dois elementos estiverem, maior a chance que a mudança em um implique na mudança em outro [4].

Quanto mais acoplado estiver o banco de dados, ou seja, dependente de diversas aplicações externas, mais difícil será a aplicação de uma refatoração [1].

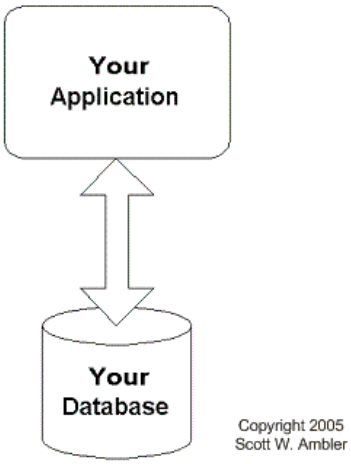

Figura 1: Baixo Acoplamento

A Figura 1 demonstra um cenário **Single-Database Application** que é bem simplificado, onde a aplicação de uma refatoração exigirá um esforço menor do que a Figura 2 onde o **Multi-Application Database** é considerado o pior caso exigindo muito planejamento e cuidado devido a dependência de inúmeras aplicações.

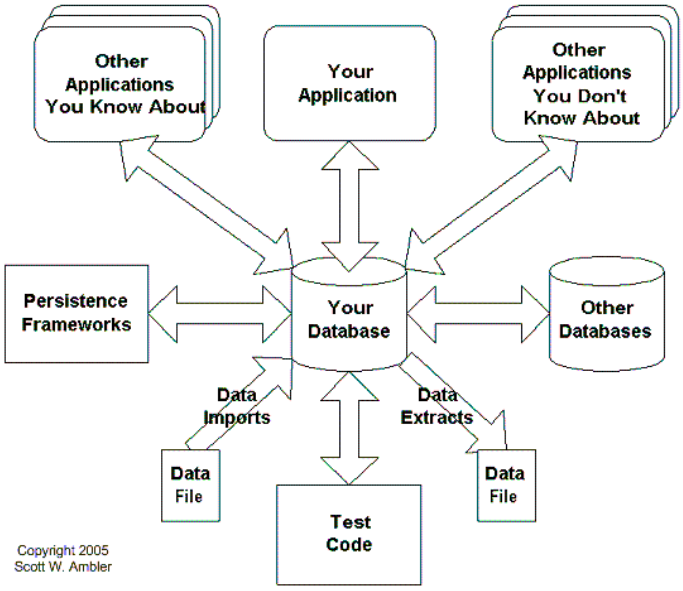

Figura 2: Alto Acoplamento

2.1.2 Processo de refatoração

Um processo é um conjunto organizado de atividades com um objetivo em comum. Executar um *database refactoring* em um cenário *Single-Database Application* ou *Multi-Application Database* requer um processo (figura 3), por mais simples que seja. A grande diferença na execução em ambos cenários é que no caso do *Multi-Application Database* o período de transição poderá ser mais longo em relação ao *Single-Database Application* [1].

Importante salientar que idealmente um *database refactoring* deve ser realizado em pequenas etapas, ou seja, cada refatoração deve ser aplicada por vez de acordo com o processo aqui descrito. A principal vantagem desta abordagem é que caso ocorra um equívoco será fácil encontrar o erro, porque será na parte da aplicação que utiliza a porção do *schema* modificada.

Conforme já citado um *database refactoring* não é uma atividade simples então caso seja identificada a real necessidade de refatorar um banco de dados então pode-se usar o seguinte roteiro (processo) como guia [1]:

1. Verificar se um *database refactoring* é apropriado: para avaliar a real necessidade da refatoração pode-se utilizar três perguntas (i) "A refatoração faz sentido?", (ii) "A mudança realmente é necessária agora? e (iii) "Vale a pena o esforço?". É comum os desenvolvedores não ter um entendimento completo a respeito do *design* do banco de dados e por este motivo é importante o DBA (*Database Administrator*) participar pois ele geralmente entende de forma mais ampla o *design* do banco de dados.

2. Escolher o *database refactoring* mais apropriado: o ponto importante aqui é compreender muito bem o problema que se quer resolver para poder ser mais assertivo na escolha do(s) *database refactoring(s)* mais apropriado(s).

3. Depreciar o esquema original do banco de dados: se diversas aplicações distintas acessam o banco de dados então é bom ter em mente que não se pode refatorar e então realizar *deploy* de todas aplicações ao mesmo tempo, então um período de transição (ou período de depreciação) se faz necessário. Neste perído tanto a versão antiga quanto a nova versão do *schema* do banco de dados são suportadas em paralelo dando tempo para os times das outras aplicações poderem ajustá-las e então efetuar *deploy* suportando apenas a nova versão do *schema* do banco de dados.

4. Testar antes, durante e após: uma abordagem como TDD (*Test-Driven Development*) ajuda a garantir que a aplicação continuará funcionando mesmo após a refatoração. E ainda novos casos de testes poderão ser adicionados para garantir a refatoração.

5. Modificar o esquema do banco de dados: nesta etapa que se escreve, ou obtém de ferramentas automatizadas, um pequeno script DDL (Data Definition Language) para aplicar a mudança. Existem boas razões para se trabalhar com pequenos scripts para refatorações individuais:

    - simplicidade: scripts de mudança pequenos e focados são mais fáceis de manter;
    - corretude: significa que os scripts de mudança poderão ser aplicados individualmente, em uma ordem apropriada, de maneira a evoluir gradualmente o *schema*;
    - versionamento: diferentes instâncias do banco de dados poderão ter diferentes versões do *schema*, então manter as pequenas mudanças versionadas facilitam a organizam a sua aplicação seja em uma instância de desenvolvimento ou mesmo em produção.

6. Migrar os dados: muitas refatorações necessitam manipular os dados de alguma maneira, algumas vezes movendo de um lugar para o outro e outras vezes apenas efetuando alguma limpeza. De forma similar a modificar o *schema* do banco de dados poderá ser necessário também escrever scripts para migrar os dados.

7. Modificar programas externos: quando o *schema* de um banco de dados muda geralmente é necessário que as aplicações que o usam também seja modificadas, ou melhor, refatoradas.

8. Executar testes de regressão: como já indicado testar cada pequena refatoração é importante para garantir que esteja funcionando, então automatizar o máximo possível ajudará na detecção de regressões quando todos os testes forem executados após aplicar uma refatoração.

9. Versionar seu trabalho: assim como código fonte os artefatos do banco de dados também devem ser versionados, dentre eles podemos destacar:

    - qualquer script criado;
    - dados de teste e/ou código para sua geração;
    - casos de teste;
    - documentação;
    - modelos.

10. Anunciar o refactoring: um banco de dados é um recurso compartilhado, então é necessário comunicar que uma refatoração está sendo realizada a todas as partes interessadas, sejam equipes internas ou externas.

Na Figura 4 é demonstrado um pequeno ciclo descrevendo um fluxo básico para aplicação de uma refatoração. Deve-se observar com atenção o **Período de Transição**, que é a fase mais importante, principalmente para cenários **Multi-Database Application** (Figura 2), onde é importante ter em mente que não geralmente é inviável aplicar a refatoração e *deploy* em ambiente de produção de todas as aplicações ao mesmo tempo.

Dificilmente consegue-se alterar todas as aplicações ao mesmo tempo, principalmente se existir dependência de fornecedores externos, então você será necessário suportar o esquema original e o esquema resultante ao mesmo tempo, para somente quando todas aplicações estiverem suportando apenas o esquema resultante, ou novo esquema, será possível depreciar antigo esquema e assim finalizar este período.

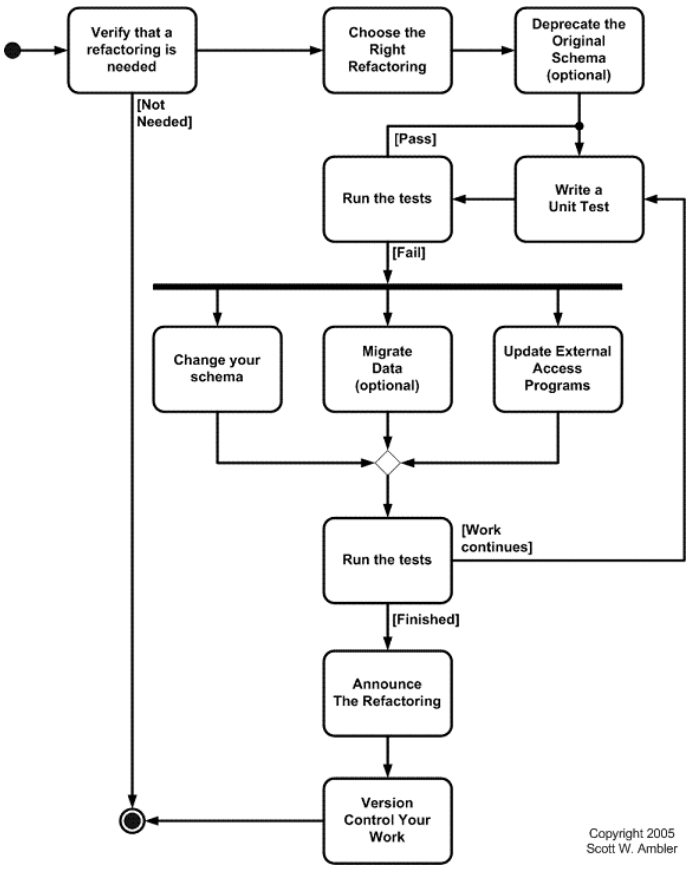

Figura 3: Processo de Refatoração de Banco de Dados

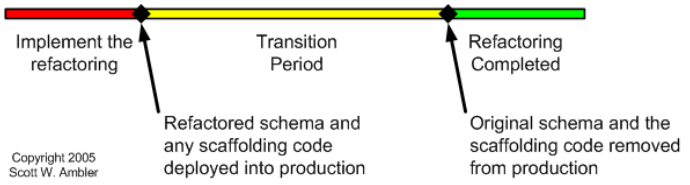

Figura 4: Ciclo de Vida de uma Refatoração de Banco de Dados

2.1.3 Estratégias de *Database Refactoring*

Existem alguns pontos a considerar como estratégias para adoção de um *database refactoring* [1]:

- Pequenas mudanças são mais fáceis de aplicar;
- Identifique unicamente cada refactoring;
- Implemente uma grande mudança realizando várias pequenas mudanças;
- Tenha uma tabela de configuração/versionamento do seu banco de dados;
- Priorize *triggers* ao invés de views ou sincronizações em lote;
- Escolha um período de transição suficiente para realizar as mudanças;
- Simplifique sua estratégia de controle de versão de banco de dados;
- Simplifique negociações com outros times;
- Encapsule acesso ao banco de dados;
- Habilite-se a montar facilmente um ambiente de banco de dados;
- Não duplique SQL;
- Coloque os ativos de banco de dados sobre controle de mudanças;

- Seja cuidadoso com políticas.

Os items acima mostram apenas algumas sugestões, em forma de **lições aprendidas**, de algumas estratégias que podem ser consideradas quando existir a necessidade de realizar uma refatoração. Para apoiar essas estratégias existe um catálogo que descrevem diversos tipos de refatorações em bancos de dados e exemplos de uso que serão apresentadas no próximo tópico deste trabalho.

2.1.4 Catálogo de *Database Refactoring*

Este catálogo é dividido em algumas categorias [1]:

1. *Structural*: são mudanças na estrutura do banco de dados (tabelas, colunas, visões, etc), como *Drop column*, *Drop table*, *Drop view*, *Introduce new column*, *Merge column*, *Split column*, entre outros.
2. *Data Quality*: são mudanças que melhoram a qualidade das informações contidas em um banco de dados, como: *Add lookup table*, *Introduce column constraint*, *Move data*, entre outros.
3. *Referential Integrity*: são mudanças que asseguram que uma linha referenciada exista em outra relação e/ou assegura que uma linha que não é mais necessária seja removida apropriadamente, como: *Add foreign key constraint*, *Add trigger for calculated column*, *Introduce soft delete*, entre outros.
4. *Architectural*: são mudanças que melhoram a maneira que programas externos interagem com a base de dados, como: *Add CRUD methods*, *Add mirror table*, *Introduce index*, *Encapsulate table with view*, entre outros.
5. *Method*: são mudanças que melhoram a qualidade de uma Procedure um Função, como: *Extract method*, *Rename method*, *Reorder parameters*, *Parametize methods*, entre outros.
6. *Transformations*: mudanças que alteram a semântica do esquema do banco pela adição de novas funcionalidades, como: *Insert data*, *Introduce new column*, *Introduce new table*, entre outros.

2.2 *Database Smells*

Fowler [2] introduziu o conceito *code smell* que é uma categoria de problemas recorrentes no código fonte que indica a necessidade de refatoração. De forma similar existem problemas recorrentes em bancos de dados que também indicam a necessidade de sua refatoração [1]. Seguem alguns exemplos de *smells*:

- *Multipurpose column*: se uma coluna for utilizada para vários fins, é provável que existe um código extra para garantir que a mesma seja usada corretamente e, muitas vezes, verificando valores de uma ou mais colunas.
- *Multipurpose table*: quando uma tabela é utilizada para armazenar vários tipos de entidades provavelmente existe uma falha de projeto.
- *Redundant data*: é um sério problema em bancos de dados, porque quando o dado é armazenado em vários locais existe alto risco de ocorrer inconsistências.
- *Tables with too many columns*: quando uma tabela tem muitas colunas é indicativo de falta de coesão, pois está armazenando dados de várias entidades.
- *Tables with too many rows*: tabelas muito grandes podem acarretar problemas de performance. O custo (memória e tempo) para buscar ou atualizar dados em uma tabela varia de acordo com o número de linhas que a mesma possui.
- *"Smart"columns*: coluna que armazena informações de mais de um contexto (concatenação de informações).

3 Trabalhos relacionados

Dois trabalhos relevantes ao tema aqui proposto foram analisados. Em Vial [5] foi apresentado um estudo de caso de um grande projeto de padronização na estrutura das bases de dados dos clientes da empresa que acabou se tornando um grande esforço de refatoração, o que os levou a desenvolver uma ferramenta de apoio a refatoração de bancos de dados.

Já as autoras D'Souza e Bhatia em [6] focaram seus esforços em criar um *Database Refactoring Framework* criando um template genérico para construção de uma ferramenta baseada nos meta-dados do banco de dados para tornar a ferramenta independente de fornecedor. A ferramenta e o trabalho não cobrem todos os *database refactorings* do catálogo 2.1.4 e as autoras concluem ao final que a ferramenta proposta ficou restrita a Oracle e MySQL por limitações do acesso aos meta-dados.

Ambos trabalhos tem uma correlação importante com este, que é no que diz respeito ao processo e as práticas de refatoração que foram discutidos em 2.1.2. Então além de reforçar a revisão de literatura realizada também o presente trabalho serve como apoio para implementação de novas ferramentas o qual é apresentado como "Trabalhos Futuros"em 5.

4 Abordagem prática de *database refactoring*

4.1 Contextualização do Problema

Dado o modelo inicial conforme figura 5 pode-se observar que a coluna *City*, que indica qual a cidade onde reside o cliente, é passível de introdução de informações incompletas e ambíguas no banco de dados, isto porque não existe qualquer validação ou verificação de consistência.

Sendo uma coluna que aceita livremente que a aplicação ou as aplicações introduzam o dado sem qualquer verificação, então responder a uma simples pergunta como ”Quais clientes residem em Santa Vitória do Palmar?”pode tornar-se uma tarefa nada trivial, pois neste cenário o nome da cidade pode ter diversas variações conforme exemplos:

- abreviado: SVP, SVDP;
- incompleto: S. Vitoria do Palmar, S. V. do Palmar, Sta Vitoria Palmar.

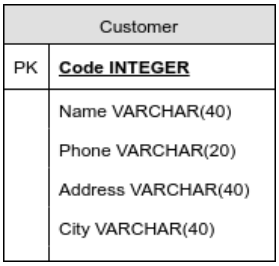

Figura 5: Modelo inicial

Um outro problema de projeto deste modelo é no que diz respeito a Normalização, pois segundo [7] uma tabela encontra-se na 1FN (Primeira Forma Normal) quando não contém tabelas aninhadas. Então serão utilizadas técnicas de refatoração de bancos de dados para efetuar a passagem a 1FN desta tabela que contém outra aninhada que neste cenário é a ”City”(cidade).

4.2 Escolha dos *refactorings* adequados e Depreciação do *schema* original

Para passagem do modelo a 1FN (Primeira Forma Normal) conforme contextualizado anteriormente precisaremos escolher alguns *refactorings* do catálogo segundo [1]. Conforme [7] a passagem para 1FN consiste em criar uma nova tabela extraindo ela da original, então seguem alguma refatorações para essa transição:

1. *Add Lookup Table*: esta refatoração consiste em criar a nova tabela chamada *City* com seus atributos *Code* e *Name*;
2. *Introduce New Column*: na tabela *Customer* adiciona-se uma nova coluna *City_Code* que será o relacionamento entre a tabela *Customer* e a nova tabela *City*
3. Add Foreign Key Constraint (Referential Integrity): relacionamento entre as tabelas *Customer* e *City* através de restrição de integridade referencial por chave estrangeira;

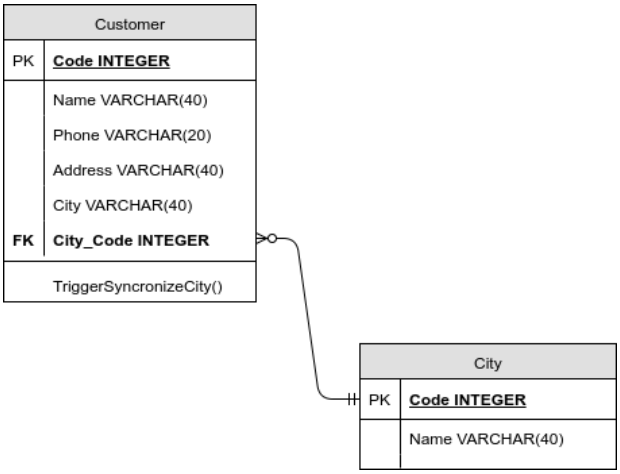

Figura 6: Período de transição

Por fim conforme apresentado na figura 6 outro elemento para garantir essa refatoração é a criação de uma *procedure* que será executada por meio de um gatinho (*trigger*) na tabela *Customer* nomeada *TriggerSynchronizeCity* que irá mapear os eventos de inserção e atualização de linhas afim de garantir a sincronização dos dados oriundos das aplicações com o novo modelo refatorado fazendo a devida relação com a nova tabela *City*. Desta forma ambos o *schemas* serão suportados e o período de transição (ou depreciação) se iniciou.

O exemplo com DDL completa para essa refatoração pode ser encontrada no arquivo sql/03_refactoring_city_transitional_period.sql no repositório [8].

4.3 Testes

Conforme já citado é extremamente testar antes, durante e após a refatoração ser realizada. Para o exemplo proposto novos testes podem ser adicionados para garantir o novo comportamento conforme exemplo:

```
CREATE OR REPLACE FUNCTION test.test_city() RETURNS VOID AS $$
-- module: city
DECLARE
  rRecord customer%ROWTYPE;
BEGIN

  -- Create dummy records
  INSERT INTO
     Customer (name, phone, address, age, photo, city)
       VALUES ('Test Customer', '1234567890', 'Test Address',
               0, 'images/photos/test_customer.png', 'Test city');

  PERFORM test.assert_not_empty(E'SELECT * FROM City WHERE description = \'TEST CITY\'');

  PERFORM test.pass();
END;
$$ LANGUAGE plpgsql;


SELECT * FROM test.run_all();
```

O caso de teste adicionado garante que a refatoração seja executada de forma correta, ou seja, ao adicionar um novo cliente com uma determinada cidade ele automaticamente através da rotina *TriggerSynchronizeCity()* irá popular a tabela *City*, então o caso de teste verifica se após essa inserção a cidade inserida no cliente existe na nova tabela criada.

Apesar deste ser um cenário mínimo para ser explorado no presente trabalho, pode-se observar que o mesmo possui características importantes como simplicidade e foco, ou seja, o caso de teste possui uma única e bem definida responsabilidade.

4.4 *Schema* resultante do banco de dados

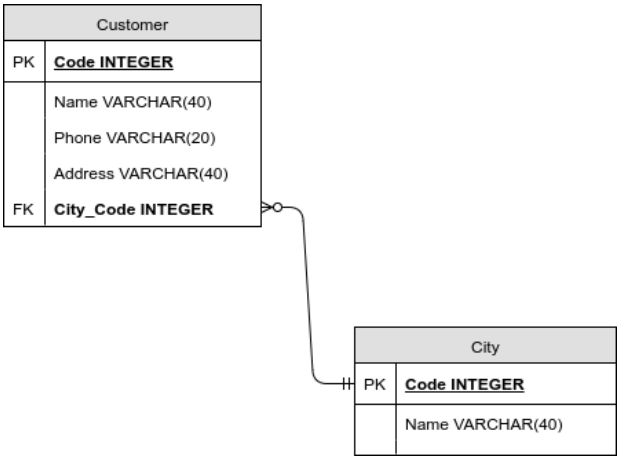

Figura 7: Modelo resultante

Na figura 7 como final do processo de refatoração temos o modelo resultante com a devida passagem a 1FN (Primeira Forma Normal) [7], desta forma como resultado da refatoração temos um modelo mais consistente, normalizado o qual minimizará introdução de dados inconsistentes ou mesmo redundantes.

Da mesma forma que no período de transição, algumas refatorações foram aplicadas para chegar neste modelo resultante:

1. *Drop procedure*: durante o período de transição se fez necessário o gatilho *TriggerSynchronizeCity()* para alimentar a nova tabela *City* com a informação do campo *City* da tabela *Customer*, suportanto assim tanto o novo quanto o antigo *schema*, portanto agora não será mais necessária esta atualização.

2. *Drop column*: esta refatoração consiste em remover a antiga coluna *City* da tabela *Customer* que mantinha o nome da cidade e agora ela está sendo armazenada na nova tabela introduzida *City*

O DDL completo desta refatoração pode ser encontrado no arquivo sql/04_refactoring_city_result.sql no repositório [8].

5 Considerações finais

Deve-se levar em consideração que apesar destas técnicas serem direcionadas para refatoração, ou seja, mudar estrutura sem mudar sua semântica, as mesmas podem e devem ser utilizadas para evolução da sua aplicação, ou seja, se existe a necessidade de construir uma nova funcionalidade em uma aplicação que está em produção, poderão ser utilizadas as práticas apresentadas neste trabalho para evoluir a estrutura de um banco de dado de forma mais consistente e segura.

O presente trabalho se limita a conceitualizar refatoração de bancos de dados relacionais e de forma prática demonstrar uma refatoração de um banco de dados relacional. O principal objetivo é ser mais uma referência guiando profissionais na aplicação de refatoração de bancos de dados.

Apesar de este trabalho apresentar um processo de refatoração, não existe qualquer limitação ao adotar apenas algumas etapas, pois todas são boas práticas de mudança em um banco de dados relacional.

Como trabalhos futuros pode-se citar:

- taxonomia dos *database smells* (primitivos e compostos)
- ferramentas de apoio:
  - detecção de *smells* através de métricas, engenharia reversa, dependência cíclica, etc;
  - apoio no processo de refatoração e até mesmo automação de algumas etapas.

Por fim, para refatorar é necessário conhecimento, disciplina, simplicidade, bom senso e persistência, sem contar no ponto fundamental que é organização. E baseado no que foi apresentado, alguns pontos a pensar em quando e porque refatorar um banco de dados: aceitar mudança de escopo, fornecer feedback rápido, melhoria contínua, aumentar simplicidade para facilitar entendimento, tornar os modelos mais próximos do mundo real, ter modelos simples para facilitar manutenção e evolução da aplicação.

Referências

[1] S. W. Ambler and P. J. Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison Wesley, 2006.

[2] M. Fowler. *Refatoração: aperfeiçoando o Projeto de Código Existente*. Bookman, 2004.

[3] F. de Royes Mello. "Database Refactoring", 2013. [http://fabriziomello.github.io/blog/2013/06/10/database-refactoring; acessado 2017-06-18].

[4] Wikipedia. "Coupling", 2005. [https://en.wikipedia.org/wiki/Coupling_(computer_programming); acessado 2017-06-18].

[5] G. Vial. "Database Refactoring Lessons from the Trenches". *IEEE Software*, vol. 32, pp. 71–79, 2015.

[6] A. D'Sousa and S. Bhatia. "Refactoring of a Database". *(IJCSIS) International Journal of Computer Science and Information Security*, vol. 6, no. 2, 2009.

[7] C. A. Heuser. *Projeto de Banco de Dados*. Editora Sagra Luzzato, 1998.

[8] F. de Royes Mello. "Repositório com exemplos práticos de Database Refactoring utilizado em um Dojo durante o Agile Brazil 2014", 2013. [https://github.com/fabriziomello/database_refactoring; acessado 2017-06-24].