Caracterizando os Problemas Enfrentados por Desenvolvedores em Aplicações Centradas em Banco de Dados

Aian Shay B. D. Cardoso
Faculdade de Computação
Universidade Federal do Pará
Belém, Brasil
aianshay@gmail.com

Gustavo Pinto
Faculdade de Computação
Universidade Federal do Pará
Belém, Brasil
gpinto@ufpa.br

***Abstract*—Uma característica de aplicações modernas é o seu foco no usuário. Mas para construir tais aplicações, os desenvolvedores precisam saber lidar com questões como acesso e manipulação aos dados, e gerenciamento do banco de dados. No entanto, essas atividades requerem habilidades específicas. Com a introdução de *frameworks* ORM, boa parte da responsabilidade com os dados pode ser abstraída. Este trabalho entrevistou 22 desenvolvedores com o objetivo de entender os problemas enfrentados por eles ao construir aplicações centradas em banco de dados. Encontramos que os problemas mais comuns são concorrência, desempenho e indexação. E que *frameworks* facilitam o trabalho de desenvolvedor mas por vezes cobram em desempenho e SQLs puros fornecem desempenho mas cobram um domínio maior de conhecimento.**

***Index Terms*—modern applications, problems, database**

I. Introdução

As aplicações que fazem uso intensivo de banco de dados constituem um dos maiores domínios de aplicação de software. Para o contexto deste trabalho, uma aplicação centrada em banco de dados é *um sistema de software que utiliza lógica guiada por banco de dados em oposição a lógica contida em programas compilados; isto é, o comportamento do software é majoritariamente ditado pelo conteúdo do banco de dados*. Em suma, aplicações de software centradas em banco de dados apoiam o desenvolvimento de funcionalidades comerciais através da manutenção de um banco de dados usando um sistema de gerenciamento de banco de dados (SGBD).

Uma outra característica de aplicações centradas em banco de dados é a capacidade destas serem independentes de tecnologia, fabricante, ou de paradigma (relacional ou não relacional). Para isso, se faz necessário tratar os SGDBs como caixas pretas, abstraindo a complexidade do acesso e manipulação aos dados, além de funcionalidades específicas.

Infelizmente, alcançar um alto grau de independência de banco de dados é um desafio para desenvolvedores de software. Isso acontece pois, na tentativa de tirar melhor proveito dos SGDBs, desenvolvedores de software muitas vezes precisam utilizar rotinas específicas de fabricante, como por exemplo escrever consultas usando PL/SQL, uma linguagem de consulta disponível somente em SGBDs Oracle.

Ademais, embora SGBDs sejam otimizados para lidar com complexas operações de gerenciamento e manipulação de dados, a forma como desenvolvedores se comunicam com o SGBD pode ter um impacto significativo na qualidade dos sistemas de software centrados em bancos de dados.

Como forma de facilitar a criação e manutenção de código que faz acesso a banco de dados, nas últimas décadas tem-se visto a introdução e popularização de um grande número de *frameworks* que realizam mapeamento objeto-relacional (do Inglês, *Object-Relational Mapping, ORM*). Frameworks ORM fornecem uma abstração conceitual entre objetos em linguagens orientadas a objetos e dados armazenados no banco de dados [12], tratando o banco de dados como objetos virtuais, que podem ser tratado no nível da linguagem de programação. Um dos principais benefícios de frameworks ORM é a facilitação do acesso e manipulação aos dados armazenados.

Através dos anos, frameworks ORM se tornaram extremamente populares. Com mais de 20 anos de histórico de desenvolvimento, o Hibernate, por exemplo, talvez o framework ORM mais conhecido e utilizado na linguagem Java (tem 4.6k estrelas e 3k forks no GitHub[1]), e hoje é um dos pilares da construção de aplicações centradas em banco de dados em Java. Por conta disto, não é surpresa ressaltar que os benefícios que frameworks ORM trouxeram ajudaram a mudar drasticamente a paisagem do desenvolvimento de software. Por exemplo, David Heinemeier Hansson, criador do framework Ruby on Rails, comentou em uma palestra na RubyConf 2018 que: "*Progress, both in terms of technology and in terms of API, allow us to move to a higher level. That higher level is where we no longer need to deal most o the time, most of our days, with SQL, even though, the relational database, is absolutely at the code of what we do*"[2]. O progresso, que David se referia, pode ser em parte atribuído ao difundido uso de frameworks ORM.

Embora os frameworks ORM tenham ajudado a drastica-

[1]https://github.com/hibernate/hibernate-orm
[2]https://youtu.be/zKyv-IGvgGE?t=741

mente reduzir a complexidade das aplicações centradas em banco de dados (e consequentemente o tempo e esforço dos seus desenvolvedores), pouco se sabe sobre quais são os desafios atuais destes desenvolvedores. Por exemplo, sabendo que é de interesse dos desenvolvedores ter independência de banco de dados (mas não é trivial alcançá-la), alguns desenvolvedores precisam criar e manter código duplicado para lidar com cada tipo de banco de dados. Ademais, frameworks ORM dificilmente são capazes de automaticamente identificar cenários de otimização, ou mesmo evitar problemas bem conhecidos, como o problema N+1, em que os dados de relacionamentos de tabelas são automaticamente consultados, aumentando o *overhead* do sistema. Para casos como esses, desenvolvedores muitas vezes precisam criar consultas SQL manuais, o que por sua vez são propensas a erros, aumentam o esforço do desenvolvimento de software e diminuem a independência de banco de dados.

Para melhor entender quais são os desafios do desenvolvedores que lidam com aplicações centradas em banco de dados, realizamos 22 entrevistas semi-estruturadas com desenvolvedores(as) e DBAs profissionais (média de $\sim$12 anos de experiência). Estes desenvolvedores trabalham desde empresas pequenas com menos de 15 funcionários, até grande corporações com mais de 6.000 funcionários. Os entrevistados trabalham com diversos sistemas escritos em linguagens como Java, C#, e JavaScript, utilizando banco de dados como PostgreSQL, MySQL, Microsoft SQL Server e MongoDB. Após análise e codificação das entrevistas, os principais achados deste trabalho foram:

1) Os principais problemas enfrentados pelos desenvolvedores com relação ao banco de dados são: concorrência, falta de conhecimento dos fundamentos de SQL, desempenho e indexação.
2) Utilizar *frameworks* para manipulação do banco de dados traz praticidade e agilidade ao desenvolvimento, mas, para realizar relacionamentos complexos ou quando o banco de dados escala, as soluções podem não ser as mais performáticas.
3) Utilizar SQL manualmente traz bastante desempenho, mas é difícil testá-lo e a barra inicial de conhecimento necessário sobre banco de dados pode ser alta. Além disso, ao utilizar manualmente, uma maior responsabilidade recaí sobre o desenvolvedor.

II. Metodologia

Nesta seção, definimos as perguntas de pesquisa e o processo utilizado para conduzir as entrevistas.

A. Perguntas de Pesquisa

Definimos cinco perguntas de pesquisa que orientaram nosso estudo. São elas:

RQ1. Quais os problemas mais comuns em aplicações que façam uso extensivo de banco de dados?

O objetivo desta primeira pergunta de pesquisa é fornecer um panorama geral das classes de problemas que são mais comuns em aplicações que fazem uso extensivo de banco de dados.

RQ2. Como os problemas nessas aplicações são detectados?

Nesta segunda pergunta de pesquisa, estamos interessados em descobrir como os problemas reportados na primeira pergunta de pesquisa são, de fato, noticiados. Por exemplo, uma abordagem para identificação destes problemas é através da escrita de testes unitários ou de integração. No entanto, estes testes são mais complexos que testes de unidade tradicionais, e portanto são menos empregados.

Esta pergunta também procura investigar se existem outras formas além dos testes utilizados para identificar os problemas encontrados.

RQ3. Qual o grau de independência que aplicações que fazem uso extensivo de banco de dados tem sobre um determinado fornecedor de banco de dados?

Por independência do banco de dados, nos referimos a possibilidade de realizar trocas entre fornecedores de banco de dados utilizado por uma aplicação, por exemplo, mudar de PostgreSQL para MySQL, de maneira simplificada. Ou de realizar até mesmo a mudança de um banco não-relacional para um relacional.

RQ4. Como desenvolvedores implementam o acesso e manipulação aos dados no banco de dados?

Na prática de desenvolvimento de software, é comum o uso de frameworks que simplificam a comunicação e acesso aos dados em um banco de dados. No entanto, aplicações que fazem uso extensivo de banco de dados podem ter necessidades específicas que não são facilmente atendidas por estes frameworks. Nessa pergunta de pesquisa, nós questionamos nossos entrevistados sobre até que ponto os frameworks de mapeamento objeto-relacionam ajudam e em que ponto estes frameworks passam a não mais apoiar o desenvolvedor.

RQ5. Quem é responsável por fazer mudanças no banco de dados?

Por fim, sabendo que aplicações que fazem uso extensivo de banco de dados precisam de otimizações específicas, na nossa última pergunta de pesquisa nós estamos interessado em descobrir quem é responsável por fazer tais mudanças.

B. Entrevistas

Nesta seção, descrevemos o procedimento realizado para selecionar os participantes para as entrevistas, e como as entrevistas foram conduzidas e analisadas.

1) Seleção dos Participantes: Optou-se inicialmente por utilizar uma abordagem de conveniência para seleção do primeiro grupo de participantes. Esse primeiro grupo foi encontrado via rede de contatos próxima. Após exaurir os potenciais contatos da nossa rede, outros contatos foram feitos em redes sociais, primeiramente no Twitter e depois no LinkedIn. Ademais, tentamos expandir a seleção dos participantes via *snow balling*; ao fim de cada entrevista, nós perguntávamos ao participante se este poderia indicar algum outro colega com o perfil indicado para nosso estudo. Quando indicado pelo

entrevistado, enviávamos um e-mail com convite para o novo candidato a entrevista.

Durante um período de quatro meses, enviamos 30 convites. Destes convites, confirmamos 22 convites com interessados em ajudar esta pesquisa. Infelizmente, alguns poucos participantes demonstraram interesse no início mas depois pararam de responder os e-mails.

Antes de realizar a primeira entrevista com um participante, realizamos uma entrevista piloto. O objetivo da entrevista piloto foi avaliar a qualidade e a duração do roteiro da entrevista. Após esta primeira entrevista, revisamos o roteiro (removemos algumas perguntas, inserimos outras novas e melhoramos algumas já existentes), por exemplo, mudamos a ordem de algumas perguntas de modo a antecipar o foco desta pesquisa - o banco de dados.

2) O Processo de Entrevista: As 22 entrevistas foram conduzidas de forma semi-estruturada. Neste tipo de entrevista, o entrevistador tem perguntas já formuladas, mas tem a liberdade de mudar ou adicionar perguntas se achar conveniente [13].

As entrevistas foram conduzidas online, utilizando o Google Meet, e foram gravadas com o consentimento prévio dos participantes. Iniciamos as entrevistas no início de Fevereiro de 2021 e terminamos no início de Junho. As 22 entrevistas foram codificadas no mês de Junho e analisadas neste mesmo mês. As 22 entrevistas duraram em média 48 minutos (mínimo: 26 minutos; máximo: 74 minutos). Baseamos as entrevistas em nossas perguntas de pesquisa RQ1–RQ5.

O roteiro da entrevista, disponível na íntegra na Seção X, foi composto de oito partes:

1) Na primeira parte, fizemos perguntas a nossos entrevistados sobre sua carreira, em particular, qual o cargo atual e quantos anos de experiencia eles possuíam, e sobre sua empresa com perguntas como qual a empresa e qual o tamanho da organização.
2) Na segunda parte, perguntamos sobre o ambiente de desenvolvimento dos entrevistados. Procuramos saber qual a pilha de tecnologias utilizada, qual o banco de dados, se usam algum modelo de desenvolvimento ágil, que tipo de interface o sistema usa (web ou aplicativo), como é feito o processo de deploy, se o modelo de desenvolvimento traz algum impacto negativo e o quão aberto o entrevistado se sente para sugerir novas tecnologias/soluções no ambiente de trabalho.
3) Na terceira parte, fizemos perguntas sobre a qualidade de software. Iniciamos perguntando os prós e contras da arquitetura utilizada, se utilizavam alguma ferramenta para avaliação de qualidade de código, se refatoração faz parte do dia-a-dia da equipe, como lidam com código duplicado e se utilizam algum *framework* para mapeamento objeto-relacional e por fim em que situações chegam a escrever SQL puro.
4) Na quarta parte, perguntamos quais os problemas mais comuns com o banco de dados, quem é responsável pelo banco e como ele enxerga a dependência em relação ao banco.
5) Na quinta parte, realizamos perguntas sobre testes. Perguntamos de que maneira a aplicação é testada e se o banco de dados é testado.
6) Na sexta parte, perguntamos sobre *bugs*. Fizemos perguntas sobre como *bugs* são identificados e qual o tipo de *bug* mais comum.
7) Na sétima parte, perguntamos sobre segurança. Perguntamos se o sistema desenvolvido utiliza algum tipo de autenticação ou autorização e como os problemas de segurança são descobertos.
8) Na última parte, para concluir, perguntamos se o entrevistado possuía experiência com sistemas que não fossem centrados em banco de dados e se ele via alguma diferença no desenvolvimento. Também deixamos o entrevistado livre caso ele desejasse comentar algo que achasse interessante/importante que por ventura não foi coberto com as perguntas feitas anteriormente.

C. Análise das Entrevistas

Nós optamos por primeiro realizar as entrevistas e só depois realizar a transcrição. Ouvimos cada gravação de cada entrevista e as transcrevemos. Não transcrevemos a entrevista completa. Em vez disso, resumimos partes relevantes das entrevistas, excluindo detalhes pequenos e ruídos sem sentido [5].

A análise das entrevistas foi composta de quatro passos:

1) Conhecendo os dados: Nesta parte, nós lemos e relemos as transcrições várias vezes para nos familiarizarmos com alguns termos específicos que ainda não conhecíamos. Quando necessário, fizemos uma breve pesquisa para conhecermos alguns termos usados pelos entrevistados.
2) Codificação inicial: Neste passo, definimos alguns códigos, isto e, rótulos que poderiam expressar o significado de partes das falas dos entrevistados. Os códigos iniciais foram considerados temporários dado que eles ainda precisariam de refinamento. Os códigos foram identificados e refinados durante toda a análise.
3) De códigos para categorias: Neste passo, já tínhamos uma lista de códigos iniciais. Então começamos a procurar por códigos semelhantes nos dados. Agrupamos os códigos com características semelhantes em categorias mais amplas. Eventualmente, nós também tivemos que refinar as categorias encontradas.
4) Refinamento de categorias: Aqui tínhamos um conjunto potencial de categorias. Então nós procuramos por evidências que apoiassem ou refutassem nossas categorias. Nós também renomeamos algumas categorias para melhor descrever os trechos de fala envolvidos nelas.

III. Descrição dos Participantes

A Tabela I mostra um resumo dos entrevistados que participaram de nosso estudo. Todos os entrevistados são brasileiros, e suas entrevistas foram realizadas em português. Nos referimos aos participantes como E1–E22. A maioria

dos entrevistados atua como desenvolvedor(a) de software. O tempo de experiência dos participantes variou entre 2 e 30 anos de experiência. Grande parte dos entrevistados trabalha em fábricas de software. O tamanho destas organizações também variou bastante, com organizações pequenas compostas por 10 pessoas até grandes, chegando a 6.000 colaboradores.

Em se tratando da pilha de tecnologias, a linguagem de programação mais usada entre os entrevistados foi Java, citada por 9 entrevistados. Em relação a banco de dados relacional, o mais utilizado pelos participantes foi o PostgreSQL (13 ocorrências); o MySQL ficou empatado com o Microsoft SQL Server com 5 ocorrências, e o banco de dados Oracle foi mencionado por 3 entrevistados. Para soluções não relacionais, a opção mais citada foi o MongoDB (citado por 7 entrevistados), comumente utilizado para para oferecer soluções de *big data*.

Alguns entrevistados optaram por utilizar tanto um banco de dados relacional quanto um não-relacional na mesma solução de software. Segundo o entrevistado E10, a necessidade de um banco relacional vem a partir do entendimento que os requisitos possuem um relacionamento muito forte entre eles. Este mesmo entrevistado também mencionou que a necessidade de um banco de dados não-relacional surgiu para atender um maior número de requisições aos dados do banco, por exemplo, para realizar análises estatísticas.

A maioria das organizações abordadas utilizava metodologias ágeis como modelo de desenvolvimento de software. A prática ágil mais citada foi o Kanban, sendo também muito comum uma simbiose desta prática com o Scrum, a segunda prática mais citada. Independente do processo de desenvolvimento, vários desenvolvedores apontaram a inclusão de praticas como reuniões diárias (*daily meetings*) e a existência de um *backlog* com as demandas a serem resolvidas pelo time.

Os sistemas em que os entrevistados trabalhavam, em sua maioria, se apresentavam ao usuário na forma de uma plataforma web (9 ocorrências). Embora em menor escala, alguns também trabalhavam com aplicativos móveis (3), APIs (3) e aplicativo *desktop* (1).

Entre os impactos negativos causados pelo modelos de desenvolvimento citados, destaca-se a demora ou atraso na entrega do produto. Este problema foi citado por quatro entrevistados, mas apenas um destes entrevistados seguia modelos ágeis (Scrum e Kanban); em todos os outros, os entrevistados descreveram um processo mais tradicional, orientado a demandas, onde um gestor define no que o time irá trabalhar ou as necessidades vem de chamados criados pelos clientes, como RUP ou RAD. Entre os modelos ágeis, uma dificuldade citada foi a adaptação inicial a cultura da empresa, que pode ser visto como um complicador para novatos que nunca trabalharam com metodologias ágeis. Além disso, também foi citado que estimar pontos de esforço no *planning poker* pode ser difícil, neste caso, o entrevistado E6 relatou que isso ocorre pois, durante a execução de uma tarefa, a equipe necessitava parar o que estava fazendo para atender pedidos de clientes maiores, e em retrospecto a pontuação inicial não condizia com o tempo que levou para executar de fato.

IV. Resultados

Nesta sessão apresentaremos nossos resultados, agrupados pelas perguntas de pesquisa.

RQ1. Quais os problemas mais comuns em aplicações que façam uso extensivo de banco de dados?

Para responder esta pergunta questionamos os entrevistados sobre quais os problemas mais comuns que eles enfrentam em relação ao banco de dados ou acesso aos dados. Obtivemos muitas respostas reportando problemas diferentes (23 diferentes no total), o que pode ser visto como um indicativo da complexidade de se criar e manter sistemas centrados em banco de dados.

Concorrência. O problema mais citado pelos entrevistados foi como tratar concorrência (7 ocorrências), tanto no sentido de fornecer acesso simultâneo a múltiplos usuários, como no contexto de prevenir problemas como *deadlock*, que acontece quando dados são requisitados por duas requisições diferentes mas nenhuma das duas tem como liberar o *lock* sobre o dado que a outra necessita: "*às vezes eu via umas queries no banco que colocavam entre parênteses no nome da tabela aquele NO LOCK e eu não entendia, até trabalhar numa empresa que fazia leilões online, aí travou no meio de um leilão e a gente teve que dar um kill na query que deu deadlock.*".

Também foi relatado o problema de concorrência no contexto de aplicações em tempo real, neste caso específico, se tratava de um jogo simulador de guerras, onde vários jogadores requisitavam vários dados ao mesmo tempo.

Algumas das soluções indicadas foram 1) a utilização de *clusters*, 2) a duplicação dos dados entre bancos diferentes e 3) rodar cada módulo do jogo sequencialmente. As soluções 1 e 2, no entanto, aumentam significativamente a complexidade da aplicação e sua manutenção, além disso, os custos envolvidos são maiores, o que é mais viável somente em empresas de maior porte.

Outro problema está relacionado a aplicações que precisam escalar para atender uma maior quantidade de usuários, mas que tem dificuldades em gerenciar esse maior número de conexões. Por exemplo, o entrevistado E7 conta que certas aplicações não contam com um *pool* de conexões para gerenciar as conexões com o banco de dados. A utilização de um *pool* mitiga problemas de sobrecarga de acesso aos dados, otimizando a quantidade de *threads* necessárias para acessar os dados: "*Outro problema comum são as aplicações que não tem um pool de conexões então ele abre uma conexão com o banco a cada requisição web, só que a conexão com o banco é um recurso caro, é boa pratica que toda aplicação tenha um pool de conexões*".

Utilização de SQL. Quase tão mencionado quanto o problema de concorrência, foi a falta de conhecimento básicos de SQL pelos desenvolvedores, mencionada por muitos participantes durante as entrevistas (6 ocorrências). De forma a reforçar esta ideia, foi citado por um dos entrevistados (que também atua como professor em uma universidade americana) que os alunos não possuíam mais uma matéria específica sobre banco

TABLE I
DEMOGRAFIA DOS ENTREVISTADOS

#	♂/ ♀	Cargo	Ramo da org.	Ling. de Prog.	SGBDs	Exp.	Tam. da org.	Duração
E1	M	Desenvolvedor	Adm. Pública	Java	Oracle, ADABAS	16	800	1h03m
E2	M	Analista de Sistemas	Pesquisa	Java, PHP	PostgreSQL, MySQL	12	1.000	1h06m
E3	M	Gerente de Projetos	Educação Superior	Java	PostgreSQL	12	54	55m
E4	M	Desenvolvedor	Fábrica de Software	Java	PostgreSQL, MongoDB	9	250	59m
E5	M	Desenvolvedor	Fábrica de Software	Java, Natural	PostgreSQL	9	6.000	44m
E6	M	Desenvolvedor Backend	Saúde	JavaScript	PostgreSQL, MongoDB	4	110	43m
E7	M	Desenvolvedor	Fábrica de Software	Java	PostgreSQL, MySQL	16	3.000	1h04m
E8	F	Desenvolvedora	Adm. Pública	C#	Microsoft SQL Server	5	100	30m
E9	M	Desenvolvedor Frontend	Fábrica de Software	Java	PostgreSQL, MongoDB	5	1.000	1h14m
E10	M	Desenvolvedor Full Stack	Jogos	JavaScript, Python	PostgreSQL, MySQL, MongoDB	5	1.100	26m
E11	M	Analista de Sistemas/CTO	Fábrica de Software	C#, VBA	Microsoft SQL Server	22	12	44m
E12	M	Empresário/Desenvolvedor	Consultoria	Python, C++	PostgreSQL	14	10	43m
E13	M	Cientista de Dados	Educação Profissional	Python	Microsoft SQL Server	5	200	29m
E14	M	Developer Advocate	Fábrica de Software	Java, PHP	MySQL, MongoDB	25	3.000	34m
E15	M	DBA	Consultoria	–	PostgreSQL	8	60	55m
E16	M	Prof. / Desenvolvedor	Educação Superior	Java, Python	PostgreSQL, Oracle	30	6.000	48m
E17	F	Desenvolvedora	Logística	Python	PostgreSQL, MongoDB	5	2.000	52m
E18	F	Analista de Testes	Finanças	–	Oracle	7	700	44m
E19	M	Arquiteto de Soluções	Finanças	JavaScript, PHP	MySQL, ArangoDB	13	80	54m
E20	M	Desenvolvedor (Líder Téc.)	Fábrica de Software	JavaScript, C#	PostgreSQL, Microsoft SQL Server	13	25	53m
E21	M	Desenvolvedor (Líder Téc.)	Fábrica de Software	C#	Microsoft SQL Server	25	200	47m
E22	F	Analista de Testes	Finanças	–	–	3	200	30m
					Média	11,98	1.177	48min
					Desvio Padrão	7,56	1.758	12,73min

de dados: " *Uma coisa que eu percebi aqui é que a galera não tá mais aprendendo banco de dados na graduação, tá surgindo esses bancos NoSQL [...] não tem as constraints do relacional. [...] Então consequentemente não sabe mais fazer SQL, alguns times que tentaram usar relacional, eles não sabiam como organizar as tabelas, eu acho que não tem mais disciplina de banco de dados, o banco tá largado lá numa disciplina de programação que o cara fala por alto. [..] Aí quando precisa o cara não sabe mais usar*". Segundo o entrevistado E12: "*Como tem muita ferramenta, a gente vive no mundo da abundância, ao longo do tempo acaba se perdendo a questão de fundamentos. Geralmente é quando precisa escalar"* (que os problemas aparecem).

Desempenho. Um terceiro problema citado frequentemente foi o desempenho das consultas no banco de dados (6 ocorrências). Um dos entrevistados (E8) mencionou que: "*o banco é grande, chega num momento que ele cria gargalos*". De acordo com o entrevistado E3, os problemas de desempenho acontecem por causa de "*grandes consultas no banco, quando precisa trazer grandes quantidades de dados*". E o entrevistado E12 adiciona: "*O carro chefe é performance - cheguei num determinado ponto que minha aplicação cresceu*".

Interpretamos esses relatos como indicativos de que um dos cenários onde problema de desempenho surge é quando o banco de dados começa a escalar, ganhando um grande número de usuários, dados e, consequentemente, requisições.

Índices. A indexação também foi um problema latente citado por vários entrevistados (4). Seja a falta de índice numa coluna muito utilizada em *queries*, índice insuficiente em um banco de dados que escalou ou uma coluna que foi indexada mas não deveria. O entrevistado E14 comentou: *"Índices e consulta mal feita, é o que mais tem. 80% dos problemas é porque as colunas não tem índice"*.

Outro entrevistado (E21) ainda citou que a falta de índice pode ocasionar lentidão: "*Você começa a sentir que as consultas estão ficando lentas, são simplesmente porque você não pensou naquela consulta no começo e não criou índices pra ela, criação de índice é uma coisa que todo mundo deveria saber e entender, ai vem novamente a base de eu entender o plano de execução de uma query"*.

Outros problemas. Outro problemas citados em menor frequência foram: falta de conhecimento do *framework*, armazenamento insuficiente, dificuldade em popular o banco para realizar testes, o clássico N + 1, atualização de versão do banco, dependência cíclica e como escalar o banco de dados.

RQ2. Como os problemas nessas aplicações são detectados?

Para responder esta pergunta, questionamos os entrevistados de que forma uma aplicação centrada em banco de dados é testada. De maneira geral, nossos entrevistados realizam testes automatizados, como o teste de unidade (8 ocorrências) e o teste de integração (5), além também do teste de regressão (3). Como processo de teste, foi citado apenas uma vez o *test-driven development* (TDD).

Mais especificamente, perguntamos também quais técnicas nossos entrevistados utilizam para realizar testes específicos para o banco de dados, por exemplo, para testar *storage procedures*, *triggers* ou testes de desempenho. No entanto, testes de *storage procedures* ou *triggers* do banco de dados não se mostraram comuns entre o nosso conjunto de desenvolvedores. Foi observado apenas um caso de um desenvolvedor (E7) que realizava testes de storage procedures. Ele comentou que este tipo de teste não é algo trivial, e como consequência foi necessário escrever uma aplicação própria em Java para este

fim. Também citou a escassez de *frameworks* para teste de banco de dados (conhecia apenas o utPLSQL) .

Quando perguntados se realizavam a implementação de regras de negócio no banco, a maioria dos entrevistados disseram que não. Um dos motivos para não transferir para o banco de dados essa responsabilidade era justamente a dificuldade em realizar testes, uma vez que torna-se necessário testar o banco de dados e a aplicação separadamente.

No contexto de testes de desempenho em banco de dados, os testes citados foram o de resiliência (o qual o banco de dados é derrubado e é analisado quanto tempo leva para o banco de dados se recuperar até funcionamento pleno) e o de carga (o qual é realizado um aumento gradativo na quantidade de requisições feitas ao banco de dados e sua estabilidade é avaliada). Também foi citado o teste de integridade, que por sua vez realiza inserções na base de dados é em seguida se verifica se os dados foram inseridos nas suas colunas correspondentes. O teste de integridade foi citado por uma analista de testes (E22) que disse inspecionar o SQL produzido pela equipe de desenvolvimento e depois analisar se a inserção foi feita corretamente no banco.

Além desses testes, foi citado também que os problemas são comumente encontrados por meio de análise de *logs*, por uma espécie de auditoria interna e em produção, quando o cliente relata algum tipo de problema. Um padrão que notamos foi que, equipes que utilizavam menos testes detectavam mais problemas em produção, reportados pelos clientes.

RQ3. Qual o grau de independência que aplicações que fazem uso extensivo de banco de dados tem sobre um determinado fornecedor de banco de dados?

Por independência do banco de dados, nos referimos a possibilidade de realizar trocas entre fornecedores de banco de dados utilizado por uma aplicação, por exemplo, mudar de PostgreSQL para MySQL, de maneira simplificada.

Parte considerável dos entrevistados (6) citou uma alta dependência em relação ao banco de dados, e que seria difícil realizar trocas entre fornecedores. Um dos entrevistados (E9) mencionou que: "*pelo simples fato de armazenar, eu já necessito disso (do banco). São poucas as aplicações que não dependem de um armazenamento de estado*". E19 mencionou que "*em algum momento a aplicação acaba se fundindo ao banco de uma forma meio simbiótica*". Outra entrevistada (E8) comentou que "*a gente não consegue pensar em mudar o banco por causa do C#*", onde a dependência ocorre pelo sistema desenvolvido estar dependente de soluções da Microsoft.

E17 mencionou que seria difícil mudar de banco pois o PostgreSQL funciona bem com dados georreferenciados, que são uma necessidade importante para o negócio.

Por fim, o entrevistado E10 reportou ainda um outro desafio para independência de banco de dados: quando se faz necessário mudar de um banco de dados não-relacional para um relacional, uma vez que exige um esforço significativo de re-modelamento dos dados (i.e., mudar de um formato não-estruturado para um estruturado). Este cenário ocorre quando dados são inicialmente apenas guardados de modo não-relacional, mas, no futuro, uma equipe descobre que é possível extrair valor dali e os dados passam a ser estruturados.

RQ4. Como desenvolvedores implementam o acesso e manipulação aos dados no banco de dados?

Em nossas entrevistas, identificamos duas principais formas de implementar mecanismos de acesso e manipulação dos dados presentes no banco de dados: 1) apoioado por *frameworks* que realizam mapeamento objeto-relacional e 2) manualmente via SQL, seja Data Definition Languange (DDL), Data Query Language (DQL) ou Data Manipulation Language (DML).

Utilização de um framework ORM. Entre os *frameworks*, o mais popular entre os entrevistados foi o Hibernate, do ecossistema Java (8 entrevistados mencionaram a utilização deste framework), seguido do Entity (5), *framework* do ecossistema .NET/C#. Dentre os motivos que levam a escolha de um framework ORM, o mais citado foi a praticidade. Por exemplo, alguns entrevistados (E10, E21) mencionaram que frameworks ORM simplificam o processo de criar entidades dentro do código (uma abordagem chamada por um dos entrevistados de *code-first*). Outros entrevistados (E19, E9) reportaram que um ganho de confiança ao usar uma ferramenta que está sendo constantemente validada e testada, não apenas no desenvolvimento mas também por muitos outros usuários. Curiosamente, todos os entrevistados que citaram praticidade como um motivador da utilização dos *frameworks* utilizavam o *Entity*. Além da praticidade, outro motivo citado, mais especificamente no contexto do Hibernate, foi a vasta documentação disponível. O entrevistado E9 mencionou que: "*(o Hibernate) é o que tem mais documentação. A melhor parte do Java é a cultura, conhecimento de 25 anos de mercado, a maturidade é maior"*.

Entre os problemas com *frameworks* está a dificuldade em fazer relacionamentos, como dito pelo entrevistado E10: *"Com o Entity, são quando entidades tem relacionamentos, ela não lida muito bem com entidades que tem relacionamentos, se forem separadas, com CRUD é muito fácil trabalhar em cima delas."* e pode ser que as queries geradas pelo *não* sejam as mais performáticas, conforme relatado por E7: "*o framework facilita muito a vida do desenvolvedor mas eles querem que você entenda o framework pra tirar proveito, caso contrario você pode gerar problemas e gargalos de performance*".

Utilização de SQL criado manualmente. Alguns entrevistados, no entanto, reportaram ainda criar SQL manualmente. Há algumas motivações para a criação de SQL manual, dentre elas: realizar consultas com vários relacionamentos e realizar cálculos mais complexos. Um exemplo citado por E21 foi: "*Quanto mais fora do seu modelo for seu relatório, ou seja, to pegando informações de mais de um tipo de entidade, menos eu recomendo o uso do ORM*", situação a qual vários entrevistados disseram que *frameworks* não lidam muito bem. Além disso, desempenho superior foi outro motivador para se criar SQLs manualmente. Esses cenários comumente envolvem uma grande volumetria de dados ou quando o

desenvolvedor antecipa que sistema computacional irá escalar rapidamente.

Conforme citado na resposta da RQ2, além de ser difícil testar SQLs, há poucos *frameworks* que auxiliam nesta tarefa. Ademais, como SQLs são tratados como Strings, as ferramenta existentes dão pouco apoio na construção e refatoração das consultas, o que torna essa atividade tediosa e propensa a erros. Além disso, criar SQLs manualmente requer que o desenvolvedor se preocupe com a limpeza dos dados de entrada que são passados como parâmetro para o comando SQL (como por exemplo, para remover caracteres ilegais como % e $ da consulta), processo esse conhecido como *data sanitize*. Nesse sentido, o entrevistado E19 comentou que: "*Eu faço [SQL] na mão, mas eu evito porque a gente fica muito desprotegido, vai na mão que você toma um SQL injection de bobeira*".

RQ5. Quem é responsável por fazer mudanças no banco de dados?

Com a popularização dos frameworks ORM, as principais responsabilidades para com o banco de dados foram assumidas pelo desenvolvedor de software. Em particular, observamos que cenários que envolvem a construção e manutenção de um banco de dados ficam mais a cargo dos próprios desenvolvedores, sendo citado por 13 entrevistados. A maioria dos desenvolvedores que disseram fazer o banco também faziam o *tuning*, embora em menor proporção. Quando a responsabilidade é passada para o desenvolvedor, notamos, no geral, um desaparecimento da figura do DBA.

Menos comum nos relatos fornecidos pelos entrevistados (4 ocorrências), foi o cenário que o profissional DBA fica responsável pelo do banco de dados. Pelas entrevistas, percebemos que dificilmente o DBA está próximo da equipe de desenvolvimento, o que burocratiza o processo de mudança no banco de dados. Por exemplo, E1 mencionou que precisa usar um sistema de *tickets* toda que que se faz necessário realizar alguma operação ou solicitar alguma mudança no banco de dados.

Observarmos um certo padrão nos dados referentes aos cenários onde existe um DBA. Trata-se, em sua maioria, de organizações públicas ou tradicionais, como instituições financeiras, que frequentemente adotam modelos de desenvolvimento mais tradicionais, como RUP ou RAD, os entrevistados E1, E3 e E21 se encaixavam neste cenário. Por outro lado, observamos que os cenários onde o desenvolvedor se destaca como o responsável pelo banco de dados são, em sua maioria, de organizações que adotam metodologias ágeis, como Scrum ou Kanban.

V. Discussão

De acordo com os resultados obtidos, conseguimos traçar duas discussões que acreditamos serem de interesse desta pesquisa:

A. Os trade-offs envolvidos da utilização de frameworks

Conforme relatado pelos entrevistados, percebemos que a camada de abstração fornecida pelo uso de *frameworks* ORM traz diversos benefícios para o desenvolvimento de sistemas centrados em banco dedados, como agilidade e facilidade de uso.

Além disso, frameworks ORM diminuem a barreira inicial para manipulação de um banco de dados, permitindo que desenvolvedores inexperientes sejam capazes de criar aplicações centradas em banco de dados, sem necessariamente terem um profundo conhecimento das tecnologias envolvidas.

Observamos ainda que os frameworks auxiliam muito bem para realização de operações básicas, como criação, leitura, atualização e deleção. No entanto, a medida que a aplicação cresce e requisitos não-funcionais como desempenho e escalabilidade se tornam mais necessários, as soluções fornecidas tendem a não apoiar o desenvolvimento adequadamente, ou então podem não ser as mais performáticas.

Como oportunidade de pesquisa, identificamos que parece ter um ponto em que as aplicações crescem e o framework não passa a ser mais suficiente para as necessidades do negócio, e o uso de SQL puro passa a ser necessário. Entender essa transição framework-SQL e em que ponto essa inflexão ocorre pode ser um bom tópico de pesquisa.

B. Os trade-offs envolvidos na utilização do SQL

Neste trabalho, alguns entrevistados reportaram que o uso de SQL criados manualmente, bem como *storage procedures* ou *triggers* podem melhorar o desempenho da aplicação centrada em banco de dados, principalmente em cenários onde a volumetria de dados é muito grande.

Porém, a utilização de SQL escritos manualmente demanda um maior conhecimento de conceitos por parte do desenvolvedor. Além disso, testar tais recursos é complicado, tanto pela escassez de ferramentas e *frameworks* para tal fim, bem como pela dificuldade a mais de ter que testar a aplicação e o banco de dados separadamente.

Por fim, ao escrever SQL, o desenvolvedor fica mais vulnerável a erros, desde erros de sintaxe até problemas de segurança, como *SQL injection*.

Em relação aos tradeoffs do SQL, vemos uma oportunidade de implementação de uma solução prática, que torne a utilização de SQL mais fácil para o desenvolvedor, levando em consideração os pontos "negativos" do SQL, como a dificuldade de testes e segurança, buscando remediar essas dificuldades. Imaginamos que esta implementação poderia ser feita na forma de um plugin para frameworks já existentes ou na forma de uma nova ferramenta separada.

C. Desenvolvedores devem ser mais DBAs?

O cenário de desenvolvimento de aplicações centradas em banco de dados que pudemos desenhar com esta pesquisa nos sugere que os desenvolvedores se tornarem cada vez mais DBAs, por meio de um maior entendimento dos conceitos que permeiam banco de dados, é uma boa pedida tanto a nível pessoal, para os profissionais, como para as organizações, que irão ter mão de obra mais apta a desenvolver soluções robustas e solucionar problemas complexos.

Se a tendência das universidades de abdicar de ensinar tais conceitos se confirmar, acreditamos que isso pode ser atingido por meio de capacitação profissional fornecida por meio das empresas, ou a nível pessoal, de forma autodidata, algo bem comum na área de tecnologia.

VI. Trabalhos relacionados

A literatura sobre a utilização de técnicas de engenharia de software para resolução de problemas em aplicações que fazem uso intensivo de banco de dados é particularmente rica.

A. Identificação de anti-padrões em restrições de SGBDs

Nijjar e Bultan [10] extraíram modelos matemáticos formais do esquema de banco de dados das aplicações Ruby on Rails e procuram por erros nos modelos. Em um trabalho seguinte, Nijjar e Bultan [11] extraíram modelos matemáticos formais do esquema de banco de dados e desenvolvem heurísticas para descobrir anti-padrões no esquema. O framework desenhado pelos autores pode então propor automaticamente soluções para corrigir o esquema do banco de dados. Gligoric e Majumdar [4] propuseram uma abordagem de verificação de modelos, chamada de DPF (Database PathFinder), para detectar problemas de concordância de bancos de dados em aplicações web. Uma limitação deste trabalho é que a abordagem proposta encontrou apenas dois problemas em 12 aplicações estudadas. Zhang et al [15] propuseram uma abordagem que utiliza análise estática para descobrir violações de restrições de integridade em aplicações centradas em bancos de dados com base em quatro anti-padrões de código, relacionados a aplicações de restrições (“Key constraint enforcement”) ou validação de dados de entrada (“exitence of input validation”) . A abordagem proposta foi utilizada em nove projetos escritos em PHP, sendo capaz de encontrar dezenas de casos dos anti-padrões de código, com alta precisão e revocação.

B. Identificação de anti-padrões em código SQL

Em 2010, Karwin lançou um livro texto compilando uma série de anti-padrões em código SQL [7], agrupados em quatro grande grupos: Logical Database Design Antipatterns, Physical Database Design Antipatterns, Query Antipatterns, and Application Development Antipatterns. A descrição destes anti-padrões é, no entanto, fortemente baseada na experiência pessoal do autor na construção de software que faz uso intensivo de banco de dados. Khumnin e Senivongse [8] conduziram um estudo com o objetivo de investigar a prevalência de cinco (dos oito anti-padrões) do grupo “Logical Database Design Antipatterns”. Para isso, os autores construíram uma ferramenta para detectar esses anti-padrões e aplicaram essa ferramenta em três projetos industriais. Foram encontradas 264 instâncias confirmadas desses smells. Foi também observado um baixo número de falso negativos (apenas 1 caso), embora um alto número de falso positivos (mais de 350 casos).

Lyu e colegas estudaram anti-padrões de código SQL e seu impacto no desempenho de aplicações móveis. Pra isso, os autores conduziram uma revisão da literatura para primeiramente categorizar o entendimento atual sobre anti-padrões em código de aplicações móveis. Após essa categorização, foi feito outro estudo para quantificar o impacto desses anti-padrões no desempenho e consumo de energia de um aplicativo bem ranqueado (disponível na plataforma Google Play). Dentre os 11 anti-padrões encontrados na revisão da literatura, foi observado que 8 destes impactaram significativamente no desempenho e consumo de energia das aplicações móveis. O estudo de Filho e colegas [3] investiga más práticas de programação em código PL/SQL (Procedural Language for SQL), uma extensão da linguagem SQL que inclui construções como laços e condicionais implementadas no banco de dados Oracle. Após uma investigação em 20 projetos do GitHub, foi observado que o smell mais recorrente é relacionado ao uso da clausula DEFAULT em declaração de parâmetros.

C. Identificação de anti-padrões em código ORM

Alguns outros trabalhos estudam eventuais práticas e desafios com o uso de frameworks que facilitam o mapeamento objeto-relacional. Como as consultas ORM não são tipadas estaticamente, Cook e Rai [2] apresentam uma abordagem para representar consultas ORM como objetos tipados estaticamente, como forma de evitar erros de runtime. O trabalho de Nazário e colegas [9] introduz um framework para identificação de outros anti-padrões, no entanto, focados em código que realiza acesso a dados por meio de um framework ORM JPA, da linguagem Java. Após conduzir um estudo da usabilidade do framework com 13 desenvolvedores profissionais, foi observado que em geral o framework ajudou os desenvolvedores a encontrar eventuais problemas de mapeamento, além de garantir que boas práticas de código sejam utilizadas.

D. Identificação de problemas de desempenho

Há alguns outros trabalhos que visam descobrir problemas de desempenho no código de acesso à base de dados usando análise de programa. Por exemplo, Smith e Williams [14] documentaram os problemas e as possíveis soluções para uma série de anti-padrões relacionados ao desempenho do banco de dados. Um dos padrões discutidos, chamado de “Empty Semi Trucks”, ocorre quando um grande número de consultas excessivas ao banco de dados (por exemplo, selecionar, inserir, atualizar ou excluir) são enviadas ao banco de dados para uma determinada tarefa. O trabalho de Hoek-stra [6] por sua vez propõe abordagens para detectar estatisticamente os anti-padrões no código de acesso ao banco de dados. Neste trabalho, no entanto, o autor não forneceu nenhuma avaliação do impacto para os problemas detectados.

Sabendo que frameworks ORM operam em um nível mais baixo (no acesso aos dados), estes frameworks dificilmente conseguem saber como os dados vão ser utilizados pelas aplicações de software. Dessa forma, frameworks ORM não conseguem otimizar o conjunto de dados que precisa ser retornado. O trabalho Tse-Hsun Chen [1] ataca esse problema através da construção de uma técnica que detecta quais dados redundantes são retornados. Diminuir os dados redundante

das transações melhorou o desempenho das aplicações em até 17%.

Nosso trabalho se diferencia destes outros principalmente pela metodologia mais exploratória, ao invés de construirmos ferramentas, com as entrevistas nós queríamos ouvir dos desenvolvedores quais os desafios que eles enfrentam em relação ao banco de dados no seu dia-a-dia.

VII. Ameaças a Validade

Como todo estudo empírico, o nosso também tem limitações e ameaças a validade.

Primeiro, todos os entrevistados são brasileiros. Embora um entrevistado trabalhe fora do país, todos os demais também atuam no Brasil. Ademais, dos 22 entrevistados, apenas três são mulheres. A falta de representatividade das mulheres é uma limitação do trabalho. Em nossos convites tentamos maximar o número de mulheres, mas tivemos pouco retorno destas mulheres.

Similarmente, embora tenhamos conduzido 22 entrevistas, nós não alcançamos saturação nos dados. Saturação acontece quando as respostas dos entrevistados começam a convergir, indicando que chegamos a um bom entendimento do problema estudado. Por exemplo, ainda observamos novos problemas na construção e manutenção de sistemas de software centrado em banco de dados. Todavia, acreditamos que os problemas reportados nesse trabalho podem servir de insumos para que outros trabalhos possam aprofundar em suas causas e eventuais soluções.

Ademais, embora tivéssemos tentado diversificar a população de entrevistados, observamos que muitos destes utilizam uma pilha similar de tecnologias para desenvolvimento de software (por exemplo, Java ou C# como linguagens de programação e PostgreSQL e e MongoDB como banco de dados). Como estas tecnologias são bem estabelecidas (existem no mercado há algumas décadas), os problemas que encontramos podem não refletir os problemas que desenvolvedores que utilizam tecnologias mais modernas tem.

Por fim, como ameaça também vemos que o tempo de experiência relatado pelos entrevistados pode não condizer com as reais habilidades do desenvolvedor, embora seja um indicativo importante. Explorar mais profundamente qual o real nível de conhecimento dos desenvolvedores, se o tempo de experiência condiz com a senioridade, pode ser de grande contribuição para este trabalho.

VIII. Conclusão

Com este trabalho explanamos que os *frameworks* agregam praticidade e agilidade ao desenvolvimento, mas em situações mais robustas, como grande volume de dados e relacionamentos complexos, ele pode deixar a desejar.

A utilização de SQLs manuais comumente agrega desempenho e permite realizar cálculos e relacionamentos mais complexos, mas irá exigir um maior conhecimento do desenvolvedor e impor uma maior responsabilidade sobre o profissional.

Cabe aos times de desenvolvimento entenderem em que parte do desenvolvimento estão e quais as necessidades do momento para escolher a melhor ferramenta e alocar o profissional mais competente de acordo com as necessidades.

A. Trabalhos Futuros

O questionário utilizado em nossas entrevistas ainda foi muito amplo, contendo várias perguntas com interesses além do banco de dados. Pode-se realizar entrevistas mais focadas no banco, com perguntas mais específicas para investigar principalmente qual a causa dos problemas que caracterizamos. Além disso, como não chegamos a um ponto de saturação, pode-se continuar as entrevistas até atingir-se tal ponto, para obter um melhor panorama dos problemas que permeiam banco de dados.

IX. Agradecimentos

Primeiramente gostaria de agradecer a minha família, em especial meu pai Daniel e minha mãe Roziane, sem eles e o suporte deles eu não seria nada e nem estaria aqui escrevendo esta tese. Estar aqui hoje me graduando é uma vitória deles também.

Em segundo lugar gostaria de agradecer a todos os bons professores que encontrei nesta jornada intelectual que é um curso em uma universidade. Em especial, obrigado ao Gustavo, por uma das melhores aulas que tive na faculdade, por me mostrar o mundo acadêmico e, o mais importante, como ciência é feita, a qual eu não poderia deixar de conhecer, por me inspirar com vários *posts* em seu blog e por aceitar fazer esta tese em tão pouco tempo, eu sinceramente não achei que seria possível.

Em terceiro gostaria de agradecer aos amigos que fiz nestes quatro anos e meio e que levarei para a vida. Vitor, Renan, Takeshi, Canavarro, Pedro e Igor, obrigado por tantas conquistas juntos, aprendizados, pelos lanches na cantina do ICEN durante o intervalo das aulas e pelas idas ao Ver-o-Pesinho, foram momentos que não esquecerei. Vocês com certeza tornaram essa jornada mais fácil e em alguns momentos acho que não teria conseguido se não fosse com vocês.

X. Apêndice

Aqui listamos todas as perguntas utilizadas no processo de entrevista:

A. Parte 1 - Demografia

- Qual o seu cargo função?
- Quanto tempo você está nesse cargo?
- Quantos anos de experiência você possui com sistemas centrados em banco de dados?
- Qual a sua empresa? Quantos colaboradores há na sua empresa?

B. Parte 2 - Ambiente de desenvolvimento

- Qual a pilha de tecnologia que você trabalha?
- Qual o banco de dados?
- Sua equipe usa algum modelo de desenvolvimento ágil?
- Se sim, pode explicar brevemente o funcionamento?
- Você pode descrever um dia normal de trabalho seu?
- Como começa o desenvolvimento de uma nova funcionalidade?
- Qual a interface do usuário que seu sistema usa? (aplicativo ou web)
- Como é feito o deploy?
- Possui ambiente de desenvolvimento, teste, produção e homologação?
- Como uma nova versão passa de um ambiente pro outro? Pull request com aprovação de outro programador?
- Você acha que o modelo de desenvolvimento utilizado pela sua equipe tem algum impacto (negativo) no produto?
- Você se sente confortável para experimentar novas tecnologias?

C. Parte 3 - Qualidade do Software

- Você conseguiria descrever os prós e contras da arquitetura utilizada?
- Focar nos contras e possivelmente investigar mais.
- Você usa alguma ferramenta para avaliação da qualidade do sistema?
- Se sim, quais métricas são avaliadas? (LCOM, LOC etc.)
- Refatoração faz parte da rotina da equipe?
- O quão fácil é refatorar o seu sistema?
- Você tem receio de refatorar?
- Como você evita código duplicado entre camadas?
- Como se separa a regra de negócio entre as camadas de forma a evitar código duplicado?
- Onde as regras de negócio são implementadas?
- Alguma no banco de dados?
- Você usa algum framework ORM? Se sim, qual a vantagem e desafios?
- Mesmo com ORM, você precisa criar consultas SQL? Para que casos?

D. Parte 4 - Banco de Dados

- Quais são os problemas mais comuns envolvendo o banco de dados ou acesso aos dados?
- Qual o processo quando o esquema do banco de dados é alterado?
- Quem possui permissão para alterá-lo?
- Quem é responsável por fazer o esquema do banco de dados?
- Quem otimiza as consultas (tuning de queries) ao banco de dados?

E. Parte 5 - Testes

- De que maneira a aplicação é testada? (Teste unitário, TDD, Selenium etc.)
- Como view, controllers e models são separados para serem testados individualmente?
- Você testa o banco de dados?
- Se sim, que tipo de problemas geralmente encontra?

F. Parte 6 - Defeitos

- Como vocês identificam bugs?
- Quais tipos de bugs são mais comuns na sua aplicação?
- Por que você acha que eles acontecem com tanta frequência?
- Como você acha que ele pode ser evitado?

G. Parte 8 - Segurança

- O seu sistema utiliza algum tipo de autenticação/autorização? Se sim, qual?
- Como os problemas de segurança do sistema são descobertos?
- Qual o processo padrão nesse caso? Logs do banco? Auditoria?

H. Parte 9 - Conclusão

- Você tem experiência implementando outro tipo de sistema que não seja centrado em banco de dados?
- Se sim, você vê alguma diferença no desenvolvimento?
- Tem algo importante/crucial sobre sistemas centrados em banco de dados que não foi explorado com as perguntas feitas, que você gostaria de falar?

REFERENCES

[1] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. *IEEE Trans. Softw. Eng.*, 42(12):1148–1161, Dec. 2016.

[2] W. R. Cook and S. Rai. Safe query objects: statically typed objects as remotely executable queries. In G. Roman, W. G. Griswold, and B. Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 97–106. ACM, 2005.

[3] F. G. de Almeida Filho, A. D. F. Martins, T. da Silva Vinuto, J. M. Monteiro, Í. P. de Sousa, J. de Castro Machado, and L. S. Rocha. Prevalence of bad smells in PL/SQL projects. In Y. Guéhéneuc, F. Khomh, and F. Sarro, editors, *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 116–121. IEEE / ACM, 2019.

[4] M. Gligoric and R. Majumdar. Model checking database applications. In N. Piterman and S. A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 549–564. Springer, 2013.

[5] E. J. Halcomb and P. M. Davidson. Is verbatim transcription of interview data always necessary? *Applied nursing research*, 19(1):38–42, 2006.

[6] M. Hoekstra. *Static source code analysis with respect to ORM performance antipatterns*. PhD thesis, Master's thesis, 2011.

[7] B. Karwin. *SQL antipatterns: avoiding the pitfalls of database programming*. Pragmatic Bookshelf, 2010.

[8] P. Khumnin and T. Senivongse. Sql antipatterns detection and database refactoring process. In *2017 18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 199–205. IEEE, 2017.

[9] M. F. C. Nazário, E. Guerra, R. Bonifácio, and G. Pinto. Detecting and reporting object-relational mapping problems: An industrial report. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2019, Porto de Galinhas, Recife, Brazil, September 19-20, 2019*, pages 1–6. IEEE, 2019.

[10] J. Nijjar and T. Bultan. Bounded verification of ruby on rails data models. In M. B. Dwyer and F. Tip, editors, *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 67–77. ACM, 2011.

[11] J. Nijjar and T. Bultan. Data model property inference and repair. In M. Pezzè and M. Harman, editors, *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 202–212. ACM, 2013.

[12] E. J. O'Neil. Object/relational mapping 2008: Hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 1351–1356, New York, NY, USA, 2008. Association for Computing Machinery.

[13] C. B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Trans. Software Eng.*, 25(4):557–572, 1999.

[14] C. U. Smith and L. G. Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *Computer Measurement Group Conference*, pages 717–725. Citeseer, 2003.

[15] H. Zhang, H. B. K. Tan, L. Zhang, X. Lin, X. Wang, C. Zhang, and H. Mei. Checking enforcement of integrity constraints in database applications based on code patterns. *J. Syst. Softw.*, 84(12):2253–2264, 2011.