

INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO  
UNIVERSIDADE DE SÃO PAULO

**SCC0240 PROJETO DE BASES DE DADOS**  
**Teach.me: sistema colaborativo de aulas particulares**

Alberto Campos Neves  
Gabriel Van Loon Bodê da Costa Dourado Fuentes Rojas  
Tamiris Fernandes Tinelli

Profa.Elaine Parros M. de Souza

São Carlos - SP

2020

## 1. Definição do projeto

O projeto em questão visa criar uma ferramenta que permita a organização de aulas particulares voltada ao público universitário. Para isso, interessados que desejam aprender ou lecionar algum conteúdo devem criar um perfil na aplicação. A partir do cadastro desse perfil, o **Usuário** — que possui foto de perfil, nome, sobrenome, e-mail, um nome de usuário único e senha — é capaz tanto de ministrar aulas (agindo como um **Instrutor**) quanto de participar de **Turmas** que receberão aulas de algum Instrutor (agindo como um **Aluno**).

Um usuário, agindo como um aluno, pode optar por participar de aulas em grupo ou de aulas particulares. Para participar de uma aula em grupo o aluno deve ou cadastrar uma nova **Turma** ou entrar/ser convidado como **Participante** em uma já existente. Cada Turma possui um nome único para a sua identificação, um título, uma descrição, uma imagem, o número de participantes, um **Líder da Turma** (sendo ele o usuário responsável pela criação da mesma), um número máximo de participantes, uma situação (que indica se ainda está aberta a novos participantes ou não) e um **Chat Interno** para discussão e tomadas de decisões entre seus membros.

Caso o Aluno deseje ter aulas particulares com apenas ele e um instrutor, não é necessário que ele cadastre uma turma na aplicação. No entanto, à nível de modelagem de dados, esse cenário é considerado como um aluno que participa de uma Turma cujo único integrante e líder é ele mesmo.

O sistema irá dispor de uma hierarquia de **Disciplinas Pré-Cadastradas**. Sendo assim, cada disciplina terá um nome único responsável por sua identificação e uma referência à sua disciplina pai (com exceção das disciplinas de ordem superior, que não possuem pais). Por exemplo, poderíamos ter a Disciplina “Exatas” cujas filhas seriam as disciplinas “Matemática”, “Química”, “Física”, etc. E a Disciplina “Matemática” teria também filhas como “Cálculo”, “Estatística”, etc.

O **Líder da turma**, então, pode consultar na aplicação os Instrutores que **oferecem** a **disciplina** desejada, podendo buscar um **Oferecimento** pelo nome da disciplina, o local onde ela deve ser oferecida, nome de instrutor, horários disponíveis ou intervalo de preço.

Ao encontrar o instrutor que mais lhe agrada, o **Líder** inicia um **Chat de Negociação** exclusivo com o mesmo, para que se inicie a negociação e troca de informações. Participa do chat apenas o **Líder da Turma**, para que tudo se resolva da forma mais simples possível, e os outros membros podem apenas assistir às trocas de mensagens.

Assim que combinam horário e local via chat, o **Líder** e o **Instrutor** podem cadastrar uma nova **Proposta** no banco de dados. Essa proposta possui informações sobre o oferecimento, aula(s) desejada(s) e um código único que a identifica junto ao par **Turma-Oferecimento**. É importante explicitar que é possível o agendamento de múltiplas aulas através de uma mesma proposta.

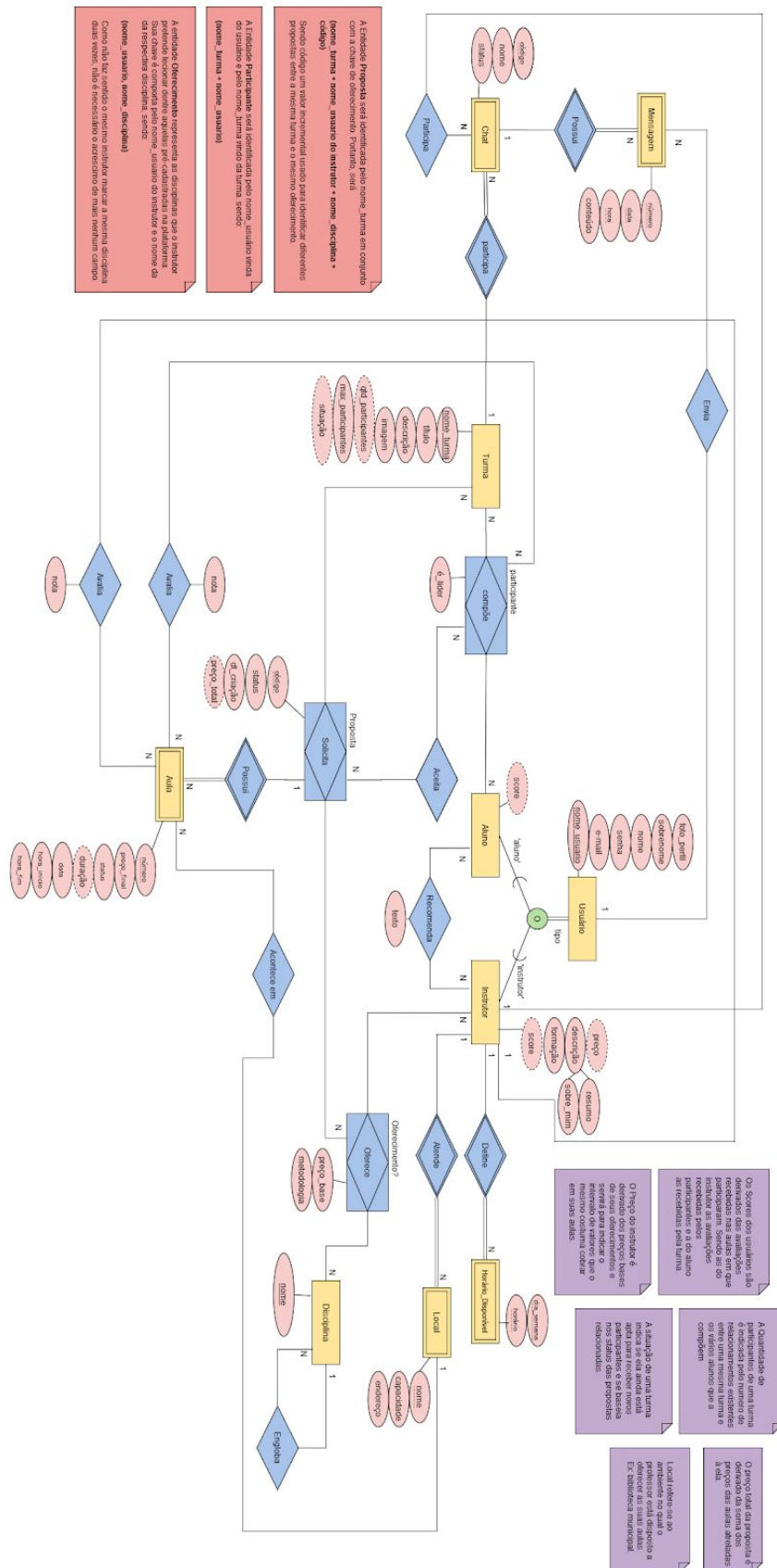
Cada **Aula** é composta pelas seguintes informações: número que identifica a aula junto à **Proposta**, duração da aula, preço final, data, hora de início e de fim e status ('agendada', 'finalizada', 'cancelada'). Além disso, cada aula está associada também a um dos **Locais** dentre as opções que o instrutor atende.

Cabe, então, ao Instrutor e à todos os **participantes** da Turma aceitarem a proposta ou não por meio de uma votação, que deve ser unânime. Essa votação deve aparecer no chat interno da turma assim que a proposta for criada no **Chat de Negociação**. Caso alguma das partes não aceite, a negociação continua e novas propostas podem ser geradas até que haja um consenso ou até que alguma das partes desista. Caso alguma proposta seja aceita, a(s) Aula(s) é(são) marcada(s).

Por outro lado, para que o Usuário, agindo como um **Instrutor**, possa ministrar aulas, é necessário o preenchimento de algumas informações adicionais no seu perfil como, por exemplo, um resumo sobre as suas habilidades, qualificações, formação, entre outros. Além disso, ele precisa cadastrar no sistema **quais Disciplinas ele Oferece** (com a descrição sobre a metodologia utilizada e preço cobrado), os **Locais** em que atende e os dias e horários da semana que costuma estar disponível para realizar atendimentos (**Horários Disponíveis**). Feito isso, tudo o que ele deve fazer é esperar por alguma solicitação proveniente de uma Turma ou de um Aluno interessado.

Por fim, após o oferecimento de uma **Aula**, Alunos e Instrutores podem fazer **Avaliações** uns aos outros. Essas avaliações possuem uma nota geral e são levadas em conta para a criação dos scores dos usuários e ranqueamento dos mesmos na plataforma.

## 2. Diagrama Entidade-Relacionamento



### 3. Consultas à base de dados

Para o bom funcionamento do projeto, o sistema deverá implementar as seguintes consultas ao banco:

- (1) Para lidar com o acesso e cadastro dos usuários ao sistema, deve ser possível
  - (a) Consultar os dados de um usuário por meio de seu **nome\_usuario**
  - (b) Consultar os dados de um instrutor por meio de seu **nome\_usuario**
  - (c) Consultar os dados de um aluno por meio de seu **nome\_usuario**
- (2) Para gerenciar as turmas de um usuário, deve ser possível
  - (a) Consultar informações das turmas que um usuário participa, indicando se o mesmo **é líder** ou não de cada uma delas
  - (b) Consultar os **participantes** de uma determinada turma
- (3) Para gerenciamento dos chats e funcionalidades atreladas deve ser possível
  - (a) Consultar os **Chats** existentes atrelados à uma determinada turma
  - (b) Consultar as **Mensagens** enviadas em um determinado chat por ordem de envio, e os respectivos **usuários** responsáveis pelas mensagens
- (4) Para a busca de um instrutor, deve ser possível utilizar os seguintes critérios (não excludentes)
  - (a) Consultar **instrutores** por meio de uma **disciplina** que ele oferece
  - (b) Consultar **Instrutores** por meio de um **horário disponível**
  - (c) Consultar **Instrutores** por meio de seu **nome** ou **nome usuário**
  - (d) Consultar **Instrutores** por meio de uma **faixa de preço**
  - (e) Consultar instrutores por um determinado **score**
- (5) Dado um instrutor, deve ser possível recuperar as seguintes informações
  - (a) Consultar **Horários Disponíveis**
  - (b) Consultar **Locais de Atendimento**
  - (c) Consultar **Oferecimentos** cadastrados pelo instrutor
- (6) Para o bom funcionamento das propostas e aulas (perspectiva aluno), deve ser possível

- (a) Consultar as **Propostas** atreladas a uma turma (de preferência permitindo agrupar ou filtrar as mesmas por meio de seus status)
  - (b) Consultar quais participantes do grupo já **aceitaram** uma dada proposta
  - (c) Dada uma proposta, consultar as **aulas** atreladas à mesma
  - (d) Dado um usuário, consultar todas as propostas de todas as turmas que o mesmo participa (idem mesmas preferências de 6.a)
  - (e) Dada uma proposta, consultar o **oferecimento** ao qual ela se refere e também o **instrutor** responsável
- (7) Para o gerenciamento das propostas e aulas (perspectiva instrutor), deve ser possível
- (a) Consultar todas as propostas atreladas à seus oferecimentos
  - (b) Consultar aulas que serão ministradas pelo instrutor, assim como as informações de data, hora, local, bem como a turma atrelada às mesmas
- (8) Por fim, para o gerenciamento de Scores de alunos e instrutores deve ser possível
- (a) Dado um aluno, consultar as avaliações de instrutores a partir de aulas das quais ele participou anteriormente
  - (b) Dado um instrutor, consultar as avaliações de participantes a partir de aulas que ministrou anteriormente

## **4. Análise dos ciclos e possíveis inconsistências**

### **1. Participante->Proposta->Aula->Participante**

É necessária a existência desse ciclo devido à necessidade em identificar o participante responsável por cada uma das avaliações de cada aula. Uma possível inconsistência seria uma aula ser avaliada por um participante de uma turma alheia, ou então, uma proposta ser aceita por um participante de outra turma. Mas novamente, este problema é evitado nas validações e regras aplicadas antes da inserção dos dados.

### **2. Disciplina->Disciplina**

Esse relacionamento é necessário para criar a estrutura de árvore de disciplinas. Aqui poderia ocorrer uma inconsistência caso uma disciplina X fosse cadastrada tanto como pai quanto filha de uma outra ou dela mesma, gerando então um ciclo. Uma opção seria modelar toda essa árvore como entidades especializadas exclusivas, isso, no entanto, seria inviável, pois limitaria a estrutura de uma maneira não desejada pela equipe.

### **3. Aluno->Turma->Chat->Mensagem->Usuário**

Este ciclo é necessário porque o chat é atrelado à uma turma, sendo ela composta por vários participantes. Um chat possui uma série de mensagens e, caso não houvesse a ligação entre a mensagem de volta para o usuário, não seria possível identificar quem foi o participante responsável pelo envio de cada mensagem. Além disso, como há chats nos quais o instrutor também participa, torna-se necessário que a mensagem seja atrelada à entidade genérica de usuário, e não diretamente aos alunos.

Uma possível inconsistência seria a de uma mensagem partir de um aluno que não participa da turma a qual o chat pertence.

### **4. Participante->Turma->Proposta->Participante**

A entidade participante é, na realidade, uma agregação da relação entre turma e aluno. Uma proposta é atrelada à turma e essa deve ser aceita pelos participantes da mesma de maneira individual. Uma possível inconsistência seria caso um participante conseguisse aceitar uma proposta de uma turma da qual não participa.

## 5. Local->Aula->Instrutor

Uma entidade Aula é vinculada a um Local do Instrutor, mas também é avaliada pelo instrutor responsável por ministrá-la. Uma possível inconsistência seria se a avaliação partisse de um instrutor diferente daquele responsável pela aula e, por consequência, diferente daquele identificado por local e horário.

## 5. Atualizações durante o planejamento da 2ª e 3ª etapa

Durante o planejamento da etapa 2, algumas decisões de projeto foram atualizadas tanto no MER quanto na documentação. Listamos a seguir as mudanças mais importantes:

- ❑ **Atualização semântica dos Horários Disponíveis:** Ficou claro que seria uma sobrecarga para o instrutor ter que cadastrar de antemão todos os horários que ele teria livre no futuro. O Horário Disponível passou a se tornar então algo mais abstrato, apenas para indicar quais horários dos dias da semana o mesmo possui livre para marcar aulas. A informação efetiva da data e hora de uma aula agora são atributos da entidade e não mais um relacionamento.
- ❑ **Atualização na Mensagem:** A mensagem agora não é mais uma entidade fraca do usuário, apenas do Chat, uma vez que é necessário manter o histórico de mensagem mesmo de pessoas que já saíram da turma ou do próprio aplicativo. Além disso, ela agora é identificada por um número inteiro crescente e não pela data e hora.
- ❑ **Atualização textual:** Alguns parágrafos foram reordenados e/ou reescritos na tentativa de deixar a definição de projeto de banco mais clara para o leitor.
- ❑ **Atualização dos ciclos:** Com as atualizações no MER, o antigo 1º ciclo foi removido, uma vez que já não há mais uma ligação entre Aula e Horário Disponível.



## 6. Modelo Relacional



## 7. Discussão e justificativas

A seguir discutimos acerca das decisões tomadas para o mapeamento do Modelo Entidade e Relacionamento para o Projeto lógico.

### Generalização Usuário e Especialização Aluno

No mapeamento da entidade Genérica Aluno e suas especializações, decidimos por unir em uma única tabela as entidades Aluno e Usuário na tabela **Usuario**.

O motivo da junção se deve ao fato de o Aluno em si não possuir nenhum atributo próprio além do “*score*”, que é um atributo derivado. A criação de uma tabela para ele, portanto, apenas aumentaria o custo de junções sem que houvesse nenhum tipo de vantagem adicional. A junção também teve embasamento semântico com o projeto, uma vez que todos os usuários cadastrados são considerados como potenciais Alunos no sistema.

A fim de prevenir possíveis confusões, ao juntar os relacionamentos de Usuário e Aluno, decidimos por nos atentar ao nomear as chaves estrangeiras como ‘Usuário’ quando fizerem referência à generalização ou como ‘Aluno’ quando fizerem referências à especialização.

**Vantagens:** economizar junções desnecessárias; confirmar que todo Usuário será também um Aluno.

**Desvantagens:** possíveis confusões para diferenciar relacionamentos da especialização e da generalização, porém facilmente resolvido ao se nomear as chaves estrangeiras corretamente.

### Generalização Usuário e Especialização Instrutor

No mapeamento da entidade Instrutor, decidimos por manter o mesmo em uma tabela separada e adicionamos uma propriedade booleana em Usuário (“*é\_instrutor*”) para indicar se um usuário possui ou não a especialização de Instrutor.

Outra alternativa seria unir a tabela Instrutor com a tabela Usuario, assim como fizemos com a tabela Aluno. Entretanto, decidimos descartar essa opção, uma vez que os atributos de Instrutor são bastante custosos (“*sobre\_mim*”, por exemplo, pode chegar a ocupar três mil caracteres, uma vez que é um texto mais elaborado) e inferimos que a maior

parte dos usuários não agiria como Instrutor. Portanto, esses campos seriam majoritariamente nulos, tornando a tabela dispendiosa.

Já quanto ao booleano, poderíamos não implementá-lo. Porém, isso tornaria necessária uma junção ou uma consulta extra entre as tabelas Usuario e Instrutor cada vez que precisássemos consultar se um usuário age como instrutor ou não. Por isso, decidimos que seria útil implementar esse atributo, apesar do custo de memória.

**Vantagens:** evitar consumo de espaço não utilizado na tabela Usuario; garantir a semântica dos relacionamentos de instrutor.

**Desvantagem:** necessidade de junção adicional de Instrutor com Usuário em algumas consultas; gasto de memória do banco com o atributo “é\_instrutor”.

### **Atributo *Endereço* da entidade Local**

No atributo “*endereço*” da entidade Local, decidimos por quebrar os dados em 4 atributos (“*endereço*”, “*complemento*”, “*cidade*”, “*uf*”) por achar que desta forma seria mais viável para o projeto de software criar sistemas de busca de forma mais eficiente, uma vez que utilizar o comparador LIKE para encontrar uma dada cidade ou estado dentro de um endereço seria mais caro do que comparações mais estritas.

Além disso, quebrar o atributo em “*cidade*” e “*uf*” e torná-los mais restritos permite que buscas e filtros sejam mais precisos e não tragam lixos em casos que a cidade e o estado possuem o mesmo nome (Ex: São Paulo e Rio de Janeiro) ou estão presentes no logradouro.

Uma outra alternativa seria criar duas novas entidades Cidade e Estado no banco e pré-cadastrar com uma lista de cidades, porém isso aumentaria o número de junções todas as vezes que o endereço fosse requisitado, além de ser necessário garantir que o usuário inserisse o nome da cidade e a sigla exatamente igual ao que se encontra no banco.

**Vantagens:** garantir um consumo menor em buscas que procuram por uma cidade ou estado específicos.

**Desvantagem:** aumenta a responsabilidade da aplicação em garantir que os dados serão inseridos corretamente.

### **Atributo ID da entidade Proposta**

O atributo artificial “ID” foi utilizado como chave primária devido ao fato da tupla (“turma”, “instrutor”, “disciplina”, “código”) ser muito grande. Como a chave de proposta é chamada por outras entidades, ficaria cada vez mais custoso se referenciar à estas entidades.

**Vantagens:** chave primária mais barata para realizar referências e constraints.

**Desvantagem:** perda do sentido semântico na chave primária; necessidade de uma nova junção nas entidades que dependem de alguma informação contida na tupla (turma, instrutor, disciplina, código).

### **Atributo Local e Instrutor da Entidade Fraca Aula**

A tupla (“local”, “instrutor”) faz referência à entidade Local. Decidimos trazer a chave inteira mesmo que seja possível recuperar o instrutor por meio de uma junção com proposta, pois percebemos que tornaria mais simples algumas consultas que seriam de interesse do Instrutor (Ex: buscar suas aulas futuras, calcular seu score, etc).

Desta forma, temos a possibilidade de uma inconsistência. Poderíamos ter um instrutor na entidade Aula que não faz parte da entidade Proposta. Decidimos manter o modelo e tratar a inconsistência a nível de aplicação, tendo em vista o ganho na otimização de certas consultas.

**Vantagem:** economia de junção em consultas de aulas de um instrutor.

**Desvantagem:** redundância da chave do instrutor; aumento no tamanho da entidade Aula.

### **Relacionamento Instrutor-avalia-Aula**

O atributo do relacionamento, “nota\_instrutor”, foi incorporado pela entidade Aula devido ao fato de o mesmo ser um dado simples e, portanto, seria mais custoso fazer uma junção para buscá-lo do que simplesmente atrelá-lo diretamente na entidade.

Além disso, caso fosse criada uma tabela para representar esta relação, a mesma teria  $\frac{3}{4}$  dos dados redundantes apenas para salvar as chaves estrangeiras (instrutor, proposta, número) e isso seria extremamente ineficiente e desnecessário.

**Vantagem:** economizar quantidade de junções; evitar repetição de dados redundantes.

**Desvantagem:** nenhuma.

### **Relacionamento Instrutor-participa-Chat**

O atributo “*instrutor*”, que faz referência ao instrutor que participa em um dado chat, foi salvo diretamente na Entidade Chat e o principal motivo foi o de diminuir a quantidade de junções na hora de fazer buscas por chats que um dado instrutor participa.

Apesar de existir a possibilidade de haver chats em que não há um instrutor (chats internos de turmas), inferimos que a proporção destes seria muito inferior, uma vez que o foco do projeto é de permitir a interação entre Alunos-Instrutores. Ponderando este fato, portanto, concluímos que vale a pena manter este formato, uma vez que a quantidade de espaço desperdiçado com nulos será ínfima.

**Vantagem:** diminuir a quantidade de junções.

**Desvantagem:** espaços desperdiçados em chats que não possuem um instrutor.

### **Atributo Situação da entidade Turma**

O atributo “*situação*” é um atributo derivado, mas decidimos mantê-lo salvo no banco de dados, uma vez que o mesmo é baseado nas propostas realizadas pela Turma. Como esta é uma informação que será muitas vezes consultada e depende de uma análise das Propostas atreladas, decidimos que seria mais barato mantê-la salva fisicamente na tabela e manipulá-la conforme necessidade.

Uma alternativa seria omitir esta informação, porém realizar uma segunda consulta todas as vezes que este dado fosse necessário.

**Vantagem:** economia na quantidade de consultas;

**Desvantagem:** aumento do tamanho da entidade Turma; necessidade de manter o atributo atualizado por meio de códigos ou triggers corretamente à nível de software.

### **Atributo Qtd\_participantes da entidade Turma**

Por uma situação semelhante ao atributo “*situação*”, o atributo “*qtd\_participantes*” será exibido muitas vezes e seria caro ter que fazer uma consulta extra todas as vezes que ele fosse necessário.

**Vantagem:** economia na quantidade de consultas;

**Desvantagem:** aumento do tamanho da entidade Turma; necessidade de manter o atributo atualizado por meio do software ou de triggers.

### **Atributo Duração da entidade Aula**

O atributo “*duração*” é um atributo derivado pertencente à entidade Aula e que decidimos não manter salvo no banco de dados, já que para obtê-lo basta uma operação muito simples: a subtração das colunas “*data\_fim*” de “*data\_início*”.

**Vantagem:** economia de espaço na tabela Aula.

**Desvantagem:** custo adicional da operação de subtração de datas em algumas consultas.

### **Atributo Score das entidades Aluno e Instrutor**

O atributo derivado “*score*” aparece tanto na entidade Aluno, quanto na entidade Instrutor. Decidimos não armazenar o dado no banco e calculá-lo quando fosse necessário.

Posteriormente, percebemos que uma outra opção seria armazenar o “*score*” no banco por meio de dois atributos: “*soma das notas*” já recebidas e “*quantidade de avaliações*”. Essa opção seria mais eficiente e barata. Porém, se implementada por si só, não permitiria o rastreo de quais aulas já foram avaliadas (sendo esse um aspecto muito importante para impedir manipulações no “*score*”).

Tendo em vista que a implementação do banco já estava avançada quando notamos isso, mantivemos a decisão inicial de manter o atributo não mapeado.

**Vantagem:** garantir que o score reflète o estado mais recente das avaliações tanto de instrutor quanto do aluno.

**Desvantagem:** aumento no custo de algumas operações que necessitam do score devido à necessidade de alguns agrupamentos e junções adicionais.

### **Atributo Preço da entidade Instrutor**

O atributo “*preço*” de Instrutor se refere, na realidade, à faixa de preços que o instrutor costuma cobrar em seus oferecimentos. Ele pode ser facilmente retornado ao averiguar o valor mínimo e máximo dos oferecimentos daquele instrutor e, portanto, decidimos mantê-lo fora do mapeamento.

Uma outra opção seria a de manter o atributo em disco a fim de economizar algumas queries. Isso, no entanto, demandaria que este campo fosse revisitado e alterado

sempre que os oferecimentos do instrutor fossem atualizados, podendo causar possíveis inconsistências caso alguma operação não o fizesse.

**Vantagem:** garantir a consistência ao verificar sempre os valores mais recentes; economizar espaço na tabela Instrutor.

**Desvantagem:** aumento do custo de algumas operações devido à necessidade de realizar agrupamentos para buscar o range de valores.

### **Atributo Preço\_total da entidade Proposta**

O atributo “*preço\_total*” é um atributo derivado, mas decidimos mantê-lo salvo fisicamente no banco de dados, uma vez que seria consultado com frequência e, portanto, aumentaria o custo das consultas caso fosse necessário fazer a junção e a soma do preço das aulas todas as vezes. Outro fator importante foi o de que as Aulas são instâncias não editáveis segundo a semântica da aplicação. Logo, o “*preço\_total*” não aumentaria a complexidade da aplicação.

**Vantagem:** ganho de desempenho; diminuição de custo de consulta.

**Desvantagem:** aumento no custo de espaço da tabela proposta.

### **Agregações Oferecimento, Proposta e Participante**

O mapeamento das três entidades acima foram feitos de maneira semelhante devido ao fato de que, em todas, os atributos presentes pertencem todos à entidade, e não ao relacionamento. Por isso, optamos por mapear as entidades e conectá-las diretamente nas vizinhas responsáveis pela sua criação.

Uma outra opção seria a de criar uma tabela de relacionamento intermediária, contendo apenas as chaves das entidades vizinhas e os atributos do relacionamento. Porém, como dito anteriormente, neste caso não haveria nenhuma vantagem, uma vez que, por não possuir nenhum atributo adicional seria apenas um aumento do custo de espaço do banco e também de junções em algumas situações.

**Vantagem:** ganho de desempenho. Redução de junções redundantes.

**Desvantagem:** nenhum.

## 8. Normalização

A seguir discutimos acerca das análises das dependências funcionais e nível de normalização atendido por cada uma das tabelas da base de dados.

### Mapeamento da entidade Usuário

Usuário = {nome\_usuario, email<sup>\*</sup>, senha<sup>\*</sup>, nome<sup>\*</sup>, sobrenome, foto\_perfil, é\_instrutor}

Dependências Funcionais

nome\_usuario -> email, senha, nome, sobrenome, foto\_perfil, é\_instrutor

email -> nome\_usuario, senha, nome, sobrenome, foto\_perfil, é\_instrutor

A tabela Usuário foi modelada seguindo a terceira forma normal genérica e também a BCNF. Temos todos os atributos monovalorados e todos os atributos não primários possuem uma dependência não transitiva com as chaves candidatas e estas são os elementos determinantes em cada uma das dependências.

### Mapeamento da entidade Instrutor

Instrutor = {nome\_usuario, resumo, sobre\_mim, formação}

Dependências Funcionais

nome\_usuario -> resumo, sobre\_mim, formação

A tabela Instrutor está na BCNF, já que temos todos os atributos monovalorados e todos os atributos não primários possuem uma dependência não transitiva com as chaves candidatas. Além disso, nas dependências, todos os elementos determinantes são chave.

### Mapeamento da entidade Disciplina

Disciplina = {nome, disciplina\_pai}

Dependências Funcionais

nome -> disciplina\_pai

O atributo disciplina\_pai, único atributo não primário da tabela, depende diretamente da chave primária completa, constituída pelo atributo nome. Sendo assim, não há possibilidade de inconsistências provocadas pela dependência funcional e a tabela Disciplina atende à BCNF.

### Mapeamento da entidade Oferecimento



Oferecimento = {instrutor, disciplina, preço\_base\*, metodologia}

Dependências Funcionais

instrutor, disciplina -> preço\_base, metodologia

A tabela Oferecimento está na BCNF, já que temos todos os atributos monovalorados e todos os atributos não primários possuem uma dependência não transitiva com as chaves candidatas. Além disso, nas dependências, todos os elementos determinantes são chave.

### Mapeamento da entidade Local

Local = {instrutor, nome, capacidade\*, endereço\*, complemento, cidade\*, uf\*}

Dependências Funcionais pré-Normalização 1FN

instrutor, nome -> capacidade, endereço, complemento, cidade, uf

Local = {instrutor, nome, capacidade\*, rua\*, número\*, bairro\*, complemento, cidade\*, uf\*}

Dependências Funcionais Após normalização 1FN

instrutor, nome -> capacidade, rua, número, bairro, complemento, cidade, uf

Nesta tabela já havíamos feito uma separação do atributo endereço contido no MER para outros atributos menores e mais semânticos. No entanto, reparamos que endereço ainda poderia ser separado em logradouro/rua, número do endereço e bairro.

Após este ajuste todos os atributos passaram a ser devidamente atômicos e monovalorados, além de sempre possuir dependência não transitiva com as chaves candidatas e ter determinantes que são chaves. Local, portanto, passou a cumprir a BCNF após os ajustes.

### Mapeamento da entidade Horário Disponível

Horário\_Disponível = {instrutor, dia\_semana, horário}

Dependências Funcionais

nenhuma

A tabela Horário\_Disponível não possui dependências funcionais entre seus atributos. Portanto, não admite inconsistências relacionadas aos dados inseridos. A tabela está de acordo com a BCNF.

### Mapeamento da entidade Turma

Turma = {nome, título\*, descrição, imagem, qtd\_participantes, max\_participantes\*, situação\*}

Dependências Funcionais

nome -> título, descrição, imagem, qtd\_participantes, max\_participantes, situação

A tabela Turma está na BCNF, já que temos todos os atributos monovalorados e todos os atributos não primários possuem uma dependência não transitiva com as chaves candidatas. Além disso, nas dependências, todos os elementos determinantes são chave.

#### Mapeamento da entidade Participante

Participante = {aluno, turma, é\_lider}

Dependências Funcionais

aluno, turma -> é\_lider

O atributo “é\_lider”, único atributo não primário da tabela, depende diretamente da chave primária completa, constituída pelos atributos aluno e turma. Sendo assim, não há possibilidade de inconsistências provocadas pela dependência funcional e a tabela Participante atende à BCNF.

#### Mapeamento da entidade Chat

Chat = {turma, código, nome\*, status\*, instrutor}

Dependências Funcionais

turma, código -> nome, status, instrutor

A tabela Chat está na BCNF, já que temos todos os atributos monovalorados e todos os atributos não primários possuem uma dependência não transitiva com as chaves candidatas. Além disso, nas dependências, todos os elementos determinantes são chave.

#### Mapeamento da entidade Mensagem

Mensagem = {turma, código, número, usuario, data\_envio\*, conteúdo\*}

Dependências Funcionais

turma, código, número -> usuario, data\_envio, conteúdo

A tabela Mensagem está na BCNF, já que temos todos os atributos monovalorados e todos os atributos não primários possuem uma dependência não transitiva com as chaves candidatas. Além disso, nas dependências, todos os elementos determinantes são chave.

#### Mapeamento da entidade Proposta

Proposta = {ID, turma\*, instrutor\*, disciplina\*, código\*, status\*, data\_criação\*, preço\_total\*}

Dependências Funcionais

ID -> turma, instrutor, disciplina, código, status, data\_criação, preço\_total

turma, instrutor, disciplina, código -> ID, status, data\_criação, preço\_total

A tabela Proposta está na BCNF, já que temos todos os atributos monovalorados e todos os atributos não primários possuem uma dependência não transitiva com as chaves candidatas. Além disso, nas dependências, todos os elementos determinantes são chave.

#### Mapeamento da entidade Aula

Aula = {proposta, número, instrutor\*, local, preço\_final\*, status\*, data\_inicio\*, data\_fim\*, nota\_instrutor}

Dependências Funcionais

proposta, número -> instrutor, local, preço\_final, status, data\_inicio, data\_fim, nota\_instrutor

A tabela Aula está na BCNF, já que temos todos os atributos monovalorados e todos os atributos não primários possuem uma dependência não transitiva com as chaves candidatas. Além disso, nas dependências, todos os elementos determinantes são chave.

#### Mapeamento do relacionamento Aceita

Aceita = {aluno, turma, proposta}

Dependências Funcionais

nenhuma

A tabela Aceita não possui dependências funcionais entre seus atributos. Portanto, não admite inconsistências relacionadas aos dados inseridos. A tabela está de acordo com a BCNF.

#### Mapeamento da entidade Avaliação Participante

Avaliação\_Participante = {aluno, turma, proposta, número, nota\*}

Dependências Funcionais

aluno, turma, proposta, número -> nota

De acordo com a semântica adotada para o banco, cada aluno pode atribuir uma nota para o instrutor a cada aula realizada.

Temos todos os atributos monovalorados e todos os atributos não primários possuem uma dependência não transitiva com as chaves candidatas. Além disso, todos os atributos determinantes são chaves. Portanto, a entidade Avaliação Participante foi modelada seguindo a terceira forma normal genérica e também a BCNF.

#### Mapeamento do relacionamento Recomenda

Recomenda = {aluno, instrutor, texto}

Dependências Funcionais

aluno, instrutor -> texto

De acordo com a semântica adotada para o banco, cada aluno pode escrever apenas um depoimento sobre sua experiência com determinado instrutor. Desta forma, aluno e instrutor determinam um texto.

Temos todos os atributos monovalorados e todos os atributos não primários possuem uma dependência não transitiva com as chaves candidatas. Além disso, todos os atributos determinantes são chaves. Portanto, a tabela Recomenda foi modelada seguindo a terceira forma normal genérica e também a BCNF.

### Conclusão

Todas as tabelas do banco de dados estão normalizadas na BCNF. Portanto, podemos afirmar que o banco também está normalizado na BCNF.

## 9. Implementação da Base de Dados e do Protótipo

Partindo agora para as implementações, vamos começar comentando acerca da implementação e dos detalhes técnicos voltados à base de dados na sessão 10 e na sessão 12 em diante iremos comentar sobre os aspectos referentes ao protótipo construído.

Os arquivos do projeto foram enviados em conjunto a este relatório e também podem ser visualizados no repositório Github dos membros do grupo.

Quanto à estrutura de pastas, os arquivos estão separados da seguinte forma:

readme.md	- Informações Gerais do projeto
/database	- Arquivos referentes à implementação da base de dados
-- setup.sql	- Arquivo para criar a base e configurar seu usuário
-- teachme_db.sql	- SQL com o script de criação das tabelas
-- mockup_data.sql	- Script para popular as tabelas
-- queries.sql	- Consultas de dificuldade média criadas para o projeto
-- conn_test.py	- Script para testar a conexão do python com a base
/api	- Protótipo de API criada com Django
/teachme-app	- Protótipo de interface em React que consome a API

O SGBD utilizado para o desenvolvimento foi o **PostgreSQL** nas seguintes especificações e ambientes:

- **Versão do SGBD testadas e utilizadas:**
  - PostgreSQL v10.12 (Gabriel)
  - PostgreSQL v12.3 (Tamiris e Alberto)

- **Sistemas Operacionais Testados**
  - Ubuntu 18.04 LTS (Gabriel)
  - Ubuntu 20.04 LTS (Alberto)
  - Windows 10 Home 1903 (Tamiris)

## 10. Configuração do Usuário e Criação do Banco

Conforme dito na seção anterior, os scripts de criação da base e estruturação das tabelas estão contidos nos arquivos “**setup.sql**” e “**teachme\_db.sql**”.

No primeiro arquivos temos apenas algumas linhas de script, como vemos a seguir, para definir o usuário que irá realizar os acessos às consultas, assim como definir o nome do banco de dados.

```
-- CRIANDO A BASE DE DADOS
DROP DATABASE IF EXISTS teachme_db;
DROP USER IF EXISTS teachme_user;

-- CRIANDO O USUÁRIO E DEFININDO PRIVILÉGIOS
CREATE USER teachme_user WITH PASSWORD 'SENHA_DO_USUARIO';
GRANT ALL PRIVILEGES ON DATABASE teachme_db TO teachme_user;
```

Já no segundo, para a criação das tabelas foi seguido o definido pelo mapeamento das entregas anteriores e, além das chaves já definidas, foram adicionados algumas CONSTRAINTS de checagem e CASCADES para aumentar ainda mais a consistência das tabelas (vide exemplo a seguir)

```
CREATE TABLE horario_disponivel(
  INSTRUTOR VARCHAR(30),
  DIA_SEMANA CHAR(3),
  HORARIO TIME,

  CONSTRAINT PK_HORARIO PRIMARY KEY (INSTRUTOR, DIA_SEMANA, HORARIO),
  CONSTRAINT FK_HORARIO_INSTRUTOR FOREIGN KEY (INSTRUTOR)
    REFERENCES instrutor (NOME_USUARIO)
    ON DELETE CASCADE
    ON UPDATE CASCADE,
  CONSTRAINT CK_DIA_SEMANA CHECK (DIA_SEMANA in ('DOM', 'SEG', 'TER', 'QUA', 'QUI', 'SEX',
'SAB'))
);
```

## 11. Consultas Levantadas Pré-criação do protótipo

Devido ao fato de grande parte das consultas imaginadas para o protótipo na primeira entrega serem relativamente simples, realizamos, logo de início, um levantamento de novas consultas mais complexas e que pudessem ser executadas em cima dos dados de teste.

Nos casos de consultas que possuíam diferentes versões, foi utilizado o comando EXPLAIN junto às consultas para realizar a análise de qual possuía uma melhor performance. A seguir mostramos então as consultas levantadas e suas respectivas implementações na versão final.

```
-- 1) Selecionar a média do preço base das disciplinas que já são oferecidas
-- por ao menos um instrutor dentro da plataforma.
SELECT O.DISCIPLINA, ROUND(AVG(O.PRECO_BASE), 2) AS MEDIA
FROM OFERECIMENTO O
GROUP BY O.DISCIPLINA;
```

**Consulta 1** - Selecionar a média do preço base das disciplinas que já são oferecidas por ao menos um instrutor dentro da plataforma.

```
-- 2) Buscar dados do usuario e, se ele for instrutor, buscar também seus dados de instrutor
SELECT U.NOME_USUARIO, U.EMAIL, U.NOME, U.SOBRENOME, I.RESUMO, I.SOBRE_MIM, I.FORMACAO
FROM USUARIO U
LEFT JOIN INSTRUTOR I
ON (U.NOME_USUARIO = I.NOME_USUARIO);
```

**Consulta 2** - Buscar dados do usuário e, se ele for instrutor, buscar também seus dados de instrutor.

```
-- 3) Selecionar, para todas as turmas existentes, a quantidade de aulas já realizadas até
-- o momento. Se uma turma ainda não realizou nenhuma aula o resultado deve ser zero.
SELECT T.NOME, COUNT(A.NUMERO)
FROM proposta P
INNER JOIN aula A ON (P.ID = A.PROPOSTA AND A.STATUS = 'FINALIZADA')
RIGHT JOIN turma T ON (P.TURMA = T.NOME)
GROUP BY T.NOME;
```

**Consulta 3** - Selecionar, para todas as turmas existentes, a quantidade de aulas já realizadas até o momento. Se uma turma ainda não realizou nenhuma aula o resultado deve ser zero.

```

-- 4) Selecionar nome_usuario, nome, nome da disciplina, preço base e metodologia dos
-- instrutores que realizam aulas em uma cidade, possuem disponibilidade no horário
-- indicado e cujo preço base da disciplina solicitada esteja dentro
-- de um intervalo de preço pré-definido. (Busca de instrutores com máximo de filtros)
SELECT DISTINCT U.NOME_USUARIO, U.NOME, O.DISCIPLINA, O.PRECO_BASE, O.METODOLOGIA
FROM oferecimento O
INNER JOIN usuario U ON (O.INSTRUTOR = U.NOME_USUARIO)
INNER JOIN local L ON (O.INSTRUTOR = L.INSTRUTOR)
INNER JOIN horario_disponivel HR ON (O.INSTRUTOR = HR.INSTRUTOR)
WHERE O.DISCIPLINA = 'Cálculo' AND O.PRECO_BASE > 0.00 AND O.PRECO_BASE < 100.00
AND L.CIDADE = 'São Carlos' AND L.UF = 'SP'
AND HR.DIA_SEMANA = 'SEG' AND HR.HORARIO = '14:00'

```

**Consulta 4** - Selecionar nome\_usuario, nome, nome da disciplina, preço base e metodologia dos instrutores que realizam aulas em uma cidade, possuem disponibilidade no horário indicado e cujo preço base da disciplina solicitada esteja dentro de um intervalo de preço pré-definido.

```

-- 5) Selecionar, para cada turma existente na base de dados, a quantidade total de propostas
-- FINALIZADAS ou APROVADAS cujo preço total das aulas foi superior ou igual à R$ 100.00.
SELECT T.NOME, COUNT(P.ID) QUANTIDADE
FROM turma T
LEFT JOIN ( SELECT P.ID, P.TURMA
FROM PROPOSTA P
WHERE P.PRECO_TOTAL >= 100.00 AND P.STATUS IN ('FINALIZADA','APROVADA')
) P ON (T.NOME = P.TURMA)
GROUP BY T.NOME;

```

**Consulta 5** - Selecionar, para cada turma existente na base de dados, a quantidade total de propostas FINALIZADAS ou APROVADAS cujo preço total das aulas foi superior ou igual à R\$ 100.00.

```

-- 6) Retornar a quantidade de horas que cada instrutor do sistema deu de
-- aula nos últimos 30 dias.
SELECT P.INSTRUTOR, SUM(A.DATA_FIM - A.DATA_INICIO)
FROM aula A
INNER JOIN proposta P ON (A.PROPOSTA = P.ID)
WHERE A.STATUS = 'FINALIZADA'
AND A.DATA_INICIO >= ('now'::timestamp - '1 month'::interval)
GROUP BY P.INSTRUTOR;

```

**Consulta 6** - Retornar a quantidade de horas que cada instrutor do sistema deu de aula nos últimos 30 dias.

```

-- 7) Retornar o username e o score dos 10 instrutores com a maior média
-- de avaliação na plataforma. Em caso de empate o desempate deve ser feito
-- pela quantidade de aulas dadas.
SELECT A.INSTRUTOR, ROUND(AVG(AP.NOTA), 2) SCORE, COUNT(DISTINCT ROW(A.PROPOSTA, A.NUMERO)) QTD
FROM aula A
INNER JOIN avaliacao_participante AP ON (A.PROPOSTA = AP.PROPOSTA AND A.NUMERO = AP.NUMERO)
GROUP BY A.INSTRUTOR
ORDER BY SCORE DESC, QTD DESC
LIMIT 10

```

**Consulta 7** - Retornar o username e o score dos 10 instrutores com a maior média de avaliação na



plataforma. Em caso de empate o desempate deve ser feito pela quantidade de aulas dadas.

```
-- 8) Para alunos que já tiveram ao menos 1 aula realizada, buscar o username do instrutor com
-- o qual ele teve mais aulas e a quantidade. Em caso de empate retornar
-- qualquer um dos possíveis resultados.
SELECT T.ALUNO, (array_agg(T.INSTRUTOR ORDER BY T.COUNT DESC))[1], MAX(T.COUNT)
FROM
  ( SELECT AC.ALUNO, A.INSTRUTOR, COUNT(*) COUNT
    FROM ACEITA AC
    INNER JOIN aula A ON (AC.PROPOSTA = A.PROPOSTA)
    WHERE A.STATUS = 'FINALIZADA'
    GROUP BY AC.ALUNO, A.INSTRUTOR
    ORDER BY COUNT DESC ) AS T
GROUP BY T.ALUNO;
```

**Consulta 8** - Para alunos que já tiveram ao menos 1 aula realizada, buscar o username do instrutor com o qual ele teve mais aulas e a quantidade. Em caso de empate retornar qualquer um dos resultados.

```
-- 9) Para cada recomendação salva no banco, retornar quantas aulas o aluno responsável pela
-- recomendação já realizou com o instrutor em questão.
SELECT R.ALUNO, R.INSTRUTOR, COUNT(*)
FROM recomenda R
INNER JOIN aceita AC ON (R.ALUNO = AC.ALUNO)
INNER JOIN proposta P ON (AC.PROPOSTA = P.ID AND R.INSTRUTOR = P.INSTRUTOR)
INNER JOIN aula A ON (A.PROPOSTA = P.ID)
WHERE A.STATUS = 'FINALIZADA'
GROUP BY R.ALUNO, R.INSTRUTOR;
```

**Consulta 9** - Para cada recomendação salva no banco, retornar quantas aulas o aluno responsável pela recomendação já realizou com o instrutor em questão.

```
-- 10) Buscar as futuras aulas já agendadas de um dado aluno, ordenando da aula mais próxima à
-- mais distante. Retornar a data da aula, o nome da turma, a disciplina da aula e o instrutor
-- responsável.
SELECT AL.DATA_INICIO, AC.TURMA, AL.INSTRUTOR, AL.PROPOSTA, AL.NUMERO
FROM AULA AL
JOIN ACEITA AC ON (AC.PROPOSTA = AL.PROPOSTA)
WHERE UPPER(AL.STATUS) = 'AGENDADA' AND AC.ALUNO = 'bob'
AND AL.DATA_INICIO > CURRENT_TIMESTAMP
ORDER BY AL.DATA_INICIO;
```

**Consulta 10** - Buscar as futuras aulas já agendadas de um aluno ordenando da aula mais próxima à mais distante. Retornar a data da aula, o nome da turma, a disciplina da aula e o instrutor responsável.



```
-- 11) Dado um instrutor, retornar a média das avaliações que ele recebeu para cada matéria que
-- ele oferece e que já recebeu ao menos 1 avaliação.
SELECT PP.DISCIPLINA, ROUND(AVG(AP.NOTA), 2) AS MEDIA_AVALIACAO
FROM AVALIACAO_PARTICIPANTE AP
JOIN PROPOSTA PP ON (PP.ID = AP.PROPOSTA)
WHERE PP.INSTRUTOR = 'ana'
GROUP BY PP.DISCIPLINA;
```

**Consulta 11** - Dado um instrutor, retornar a média das avaliações que ele recebeu para cada disciplina que ele oferece e que já recebeu ao menos 1 avaliação.

```
-- 12) Listar todas as turmas do sistema que não obtiveram NENHUMA avaliação abaixo de 3 de seus
-- instrutores nas aulas realizadas. Aulas não avaliadas pelo instrutor devem ser ignoradas.
SELECT PP.TURMA
FROM PROPOSTA PP
JOIN AULA AL ON (AL.PROPOSTA = PP.ID)
WHERE PP.STATUS IN ('APROVADA', 'FINALIZADA')
AND AL.NOTA_INSTRUTOR IS NOT NULL
GROUP BY PP.TURMA
HAVING MIN (AL.NOTA_INSTRUTOR) > 3
```

**Consulta 12** - Listar todas as turmas do sistema que não obtiveram NENHUMA avaliação abaixo de 3 de seus instrutores nas aulas realizadas. Aulas não avaliadas pelo instrutor devem ser ignoradas.

```
-- 13) Exibir o instrutor que realizou a maior quantidade de aulas no último mês em cada
-- estado. Em caso de empate exibir qualquer um dos possíveis resultados.
SELECT TODOS.UF, (array_agg(TODOS.INSTRUTOR))[1], MAX (TODOS.N_AULAS)
FROM (
    SELECT AL.INSTRUTOR, L.UF, COUNT (AL.NUMERO) AS N_AULAS
    FROM LOCAL L
    JOIN AULA AL ON (L.INSTRUTOR = AL.INSTRUTOR AND L.NOME = AL.LOCAL)
    WHERE (AL.STATUS = 'FINALIZADA')
    AND AL.DATA_INICIO >= ('now'::timestamp - '1 month'::interval)
    GROUP BY AL.INSTRUTOR, L.UF
    ORDER BY L.UF, COUNT (AL.NUMERO) DESC )
AS TODOS
GROUP BY TODOS.UF
```

**Consulta 13** - Exibir o instrutor que realizou a maior quantidade de aulas no último mês em cada estado. Em caso de empate exibir qualquer um dos possíveis resultados.

```
-- 14) Dado um aluno no banco, selecionar as aulas que ele já participou mas ainda não avaliou.
SELECT PR.TURMA, PR.INSTRUTOR, PR.DISCIPLINA, AU.NUMERO AS NUMERO_AULA
FROM aceita AC
INNER JOIN proposta PR ON (AC.PROPOSTA = PR.ID AND PR.STATUS IN ('APROVADA', 'FINALIZADA'))
INNER JOIN aula AU ON (AU.PROPOSTA = AC.PROPOSTA AND AU.STATUS = 'FINALIZADA')
LEFT JOIN avaliacao_participante AP ON (AP.ALUNO = AC.ALUNO AND AP.TURMA = AC.TURMA
AND AP.PROPOSTA = AU.PROPOSTA AND AP.NUMERO = AU.NUMERO)
WHERE AC.ALUNO = 'felipe' AND AP.NOTA IS NULL;
```

**Consulta 14** - Dado um aluno no banco, selecionar as aulas que ele já participou mas ainda não avaliou.

```

-- 15) Dado um aluno no banco, selecionar os instrutores cujas aulas foram
-- avaliadas com nota 4 ou superior pelo aluno, mas ainda não foram recomendados pelo mesmo.
SELECT DISTINCT AU.INSTRUTOR
FROM aceita AC
INNER JOIN aula AU ON (AU.PROPOSTA = AC.PROPOSTA)
INNER JOIN avaliacao_participante AP ON (AP.ALUNO = AC.ALUNO AND AP.TURMA = AC.TURMA
AND AP.PROPOSTA = AU.PROPOSTA AND AP.NUMERO = AU.NUMERO)
LEFT JOIN recomenda RE ON (RE.ALUNO = AC.ALUNO AND RE.INSTRUTOR = AU.INSTRUTOR)
WHERE AC.ALUNO = 'enrique' AND RE.ALUNO IS NULL AND AP.NOTA >= 4;

```

**Consulta 15** - Dado um aluno no banco, selecionar os instrutores cujas aulas foram avaliadas com nota 4 ou superior pelo aluno, mas não foram recomendados pelo mesmo.

```

-- 16) Exibir a média global de mensagens trocadas entre instrutores e líderes de turma
-- em chats de negociação até a realização da primeira iteração de proposta na aplicação
SELECT AVG(CONTAGEM.COUNT) AS MSG_ANTES_PROPOSTA
FROM (SELECT P.TURMA, P.INSTRUTOR, COUNT(*)
FROM chat C
INNER JOIN (SELECT P.TURMA, P.INSTRUTOR, MIN(P.DATA_CRIACAO)
FROM proposta P
GROUP BY P.TURMA, P.INSTRUTOR
) P ON (P.TURMA = C.TURMA AND P.INSTRUTOR = C.INSTRUTOR)
INNER JOIN mensagem M ON (C.TURMA = M.TURMA AND C.CODIGO = M.CODIGO)
WHERE C.INSTRUTOR IS NOT NULL AND M.DATA_ENVIO <= P.MIN
GROUP BY P.TURMA, P.INSTRUTOR
) CONTAGEM;

```

**Consulta 16** - Exibir a média global de mensagens trocadas entre instrutores e líderes de turma em chats de negociação até a realização da primeira iteração de proposta na aplicação

```

-- 17) Selecionar todos os instrutores que já deram aulas de TODAS as disciplinas
-- filhas de uma disciplina pré-especificada (neste caso 'Computação')
-- (Exemplo de Divisão)
SELECT DISTINCT O.INSTRUTOR
FROM oferecimento O
WHERE O.INSTRUTOR not in (
SELECT resto.INSTRUTOR FROM (
-- Todas combinações possíveis dos Instrutores com TODAS disciplinas filhas de 'Computação'
(SELECT sp.INSTRUTOR , p.DISCIPLINA
FROM (select D.NOME DISCIPLINA from disciplina D
WHERE D.DISCIPLINA_PAI = 'Computação') as p
CROSS JOIN (select distinct O.INSTRUTOR from oferecimento O) as sp
)
EXCEPT -- Operação MINUS de conjuntos
-- Combinações existentes de instrutores e disciplina
(SELECT O.INSTRUTOR , O.DISCIPLINA FROM oferecimento O, disciplina D WHERE
(O.DISCIPLINA = D.NOME AND D.DISCIPLINA_PAI = 'Computação'))
) AS resto
);

```

**Consulta 17** - Selecionar todos os instrutores que já deram aulas de TODAS as disciplinas filhas de uma disciplina pré-especificada (neste caso 'Computação'). (Exemplo de Divisão)

## 12. Implementação do Protótipo do Sistema

Vamos analisar agora o protótipo implementado para demonstrar a integração e uso de uma aplicação em conjunto com uma base de dados. O projeto foi desenvolvido dividindo-se a implementação em duas partes: uma API back-end responsável por se comunicar com o banco de dados e definir alguns endpoints; uma aplicação front-end responsável por exibir uma interface agradável e intuitiva mesmo para usuários leigos.

### Aplicação Back-end

A aplicação back-end foi implementada em **Python v3.7.1** utilizando o framework **Django v3.0.8** para servir como servidor HTTP e em conjunto com a biblioteca **Psycopg2** responsável pela conexão e execução das consultas na base de dados.

A estrutura dos arquivos e pastas se encontra dentro da pasta '/api' e foi estruturada da seguinte maneira:

```
manage.py - Inicializar o sistema: $ python3 manage.py runserver
/src
| -- settings.py - Definições e variáveis globais do Django
| -- urls.py     - Mapeiam as URL's aos devidos Controllers e Métodos
| -- /controllers - Recebem as requisições e executam os Models
| -- /models     - Responsável pelas regras de negócio e validações
| -- /dao        - Responsável pelas conexões e execução das queries
| -- /entities   - Classes que representam as tabelas do banco
| -- /libs       - Funções utilitárias
```

Fugindo um pouco do contexto de banco de dados, mas apenas para garantir um bom entendimento do papel de cada um dos arquivos. Segue abaixo um esquema genérico do funcionamento da API.

1. **Requisição chega no servidor:** o Django verifica urls.py para decidir qual Controller deve lidar com a requisição
2. **Controller:** o controller verifica se o método utilizado é válido e encaminha os parâmetros recebidos para o devido Model.
3. **Model:** realiza a validação e qualquer outro tipo de processamento necessário. Caso tudo esteja correto encaminha os dados para as classes de DAO.

4. **Data Access Object:** formata a consulta de acordo com os dados recebidos, abre a conexão e executa suas ações na base de dados.

Para todas as entidades foram implementadas 3 funcionalidades básicas: inserção, atualização e busca pela chave primária (salvo algumas exceções). No entanto, a seguir vamos mostrar apenas aquelas que foram feitas para funcionarem em conjunto com o protótipo visual.

### Aplicação Front-end

A aplicação Front-end foi criada utilizando o framework **ReactJS** em conjunto com o gerenciador de pacotes **NPM** e, portanto, todas as suas dependências estão devidamente apontadas no arquivo `package.json`.

Conforme todo projeto React, o aplicativo é dividido em componentes localizados dentro da pasta `src` e sua estrutura principal é descrita a seguir

<code>package.json</code>	- Lista de dependências do projeto
<code>/public</code>	- Alguns arquivos padrões do framework
<code>/src</code>	- Pasta com os componentes
<code>  -- App.js</code>	- Componente que inicializa a aplicação

Por fim, o projeto pode ser inicializado após o download das dependências com o comando `$ npm install` e `'npm start'`.

## 13. Funcionalidades Implementadas

Vamos focar agora nos códigos que são executados durante o funcionamento do protótipo. Para cada uma das funcionalidades iremos indicar qual a devida URL sendo chamada pela aplicação front-end quanto a api.

Logo em seguida iremos exibir a tela criada para o protótipo, uma descrição da funcionalidade e os devidos códigos que são executados na base de dados.

Todas as consultas estão explícitas no código e foram parametrizadas a fim de prevenir injeções de SQL nos parâmetros.

## Realizar Login

**API End-point:** http://localhost/api/login

**Front-end:** http://localhost/login

**Tela do protótipo:**

**Descrição:** nesta funcionalidade o usuário envia seus dados de login (username e senha) e o sistema retorna, caso eles estejam corretos, seus atributos de usuário (com exceção da senha).

**Consulta Realizada:**

```
# UserDAO():
def login(self, user:User):
    ...
    query = '''SELECT NOME_USUARIO, EMAIL, NOME, SOBRENOME, E_INSTRUTOR
               FROM usuario
               WHERE NOME_USUARIO = %s AND SENHA = %s LIMIT 1;'''
    self.cur.execute(query, [user.username, user.password])
```

## Criar Cadastro

**API End-point:** http://localhost/api/register

**Front-end:** http://localhost/criar-conta

**Tela do protótipo:**



O protótipo de tela de cadastro, intitulado "Cadastro", apresenta dois grupos de campos de entrada. O primeiro grupo, "Dados pessoais", contém campos para "Nome" e "Sobrenome". O segundo grupo, "Dados de acesso", contém campos para "E-mail", "Nome de usuário" e "Senha".

**Descrição:** nesta tela o usuário pode criar uma conta inserindo os seus dados e enviando o formulário. Caso ele também queira se cadastrar como instrutor basta checar um campo e inserir também alguns dados adicionais.

**Consultas Realizada:**

```
# UserDAO():
def insert(self, user:User):
    ...
    self.connect()
    query = '''INSERT INTO usuario (NOME_USUARIO, EMAIL, SENHA, NOME, SOBRENOME, E_INSTRUTOR)
              VALUES (%s, %s, %s, %s, %s, %s);'''

    self.cur.execute(query, [user.username, user.email, user.password,
                             user.name, user.last_name, user.is_instructor])
```

```
# InstructorDAO():
def insert(self, instructor:Instructor):
    ...
    self.connect()
    query = '''INSERT INTO instrutor (NOME_USUARIO, RESUMO, SOBRE_MIM, FORMACAO)
              VALUES (%s, %s, %s, %s);'''

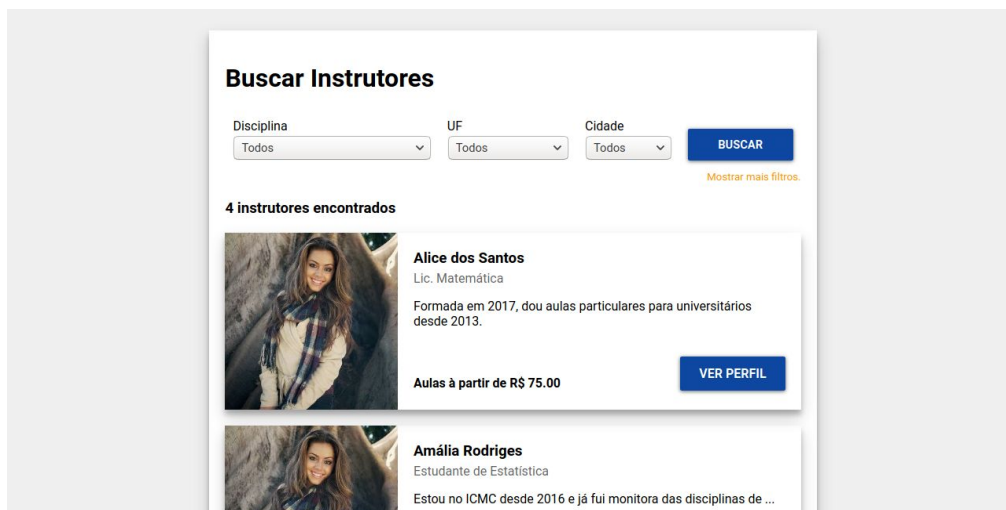
    self.cur.execute(query, [instructor.username, instructor.abstract,
                             instructor.about_me, instructor.degree])
    self.con.commit()
```

## Buscar Instrutores

**API End-point:** http://localhost/api/instructors

**Front-end:** http://localhost/instrutor

**Tela do protótipo:**



**Descrição:** Nesta tela o usuário pode visualizar os instrutores já cadastrados na plataforma que oferecem ao menos 1 disciplina. Há uma gama de filtros possíveis que modificam drasticamente a consulta (sendo essa uma das mais complicadas do protótipo). Ela pode ser vista como uma variação da consulta vista na sessão 11 - Consulta 4.

### Consulta Realizada com Todos os filtros

```
# InstructorDAO():
def get_instructors(self, subject='', city='', state='', weekday='', time='', max_price=''):
    ...
    # Montando as colunas:
    query = ''' SELECT DISTINCT ON (U.NOME_USUARIO)
                U.NOME_USUARIO, U.NOME, U.SOBRENOME, I.FORMACAO, I.RESUMO,
                O.DISCIPLINA, O.PRECO_BASE

                FROM oferecimento O
                INNER JOIN usuario U ON (O.INSTRUTOR = U.NOME_USUARIO)
                INNER JOIN instrutor I ON (O.INSTRUTOR = I.NOME_USUARIO)
                INNER JOIN local L ON (O.INSTRUTOR = L.INSTRUTOR)
                INNER JOIN horario_disponivel HR ON (O.INSTRUTOR = HR.INSTRUTOR)

                WHERE O.DISCIPLINA = %s AND L.UF = %s AND L.CIDADE = %s AND
                       HR.DIA_SEMANA = %s AND HR.HORARIO = %s O.PRECO_BASE <= %s

                ORDER BY U.NOME_USUARIO ASC, O.PRECO_BASE ASC;'''
```

**Obs:** acima se vê a query que é executada em conjunto com todos os filtros. O código, no entanto, adapta a consulta conforme os parâmetros requisitados pelo usuário a fim de otimizar a performance da mesma.

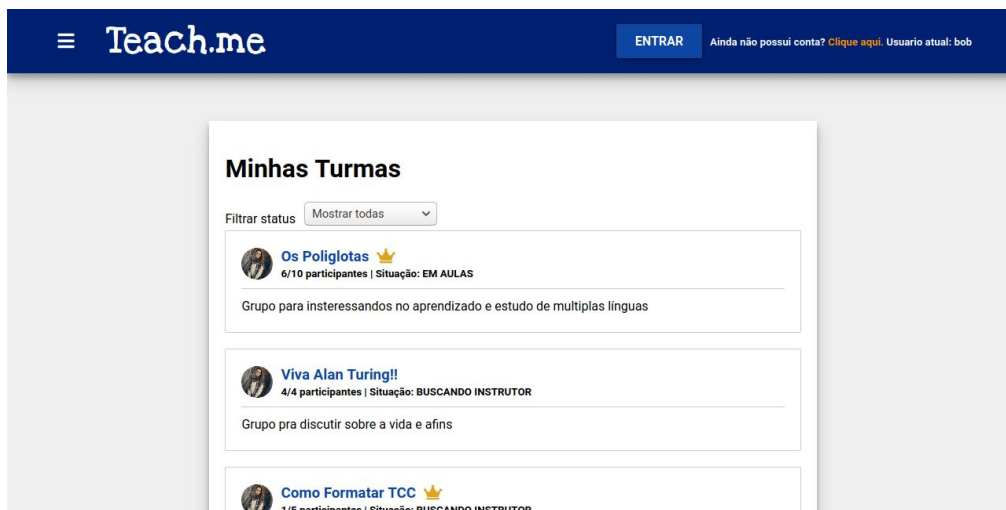


## Minhas Turmas

**API End-point:** http://localhost/api/my-classes

**Front-end:** http://localhost/painel/minhas-turmas

**Tela do protótipo:**



**Descrição:** Nesta tela o sistema solicita que a api envie todas as turmas que o usuário participa, indicando em quais delas o mesmo é líder ou não. Também é possível utilizar um filtro que busca apenas as turmas em uma determinada situação.

**Consulta Realizada:**

```
# ClassDAO():
def get_classes(self, username='', situation=''):
    ...
    query = '''SELECT T.NOME, T.TITULO, T.DESCRICAO, T.QTD_PARTICIPANTES, T.MAX_PARTICIPANTES,
                T.SITUACAO, P.E_LIDER
                FROM turma T
                INNER JOIN participante P ON (T.NOME = P.TURMA)
                WHERE P.ALUNO = %s'''
    if (situation != ''):
        query += ' AND T.SITUACAO = %s'
```

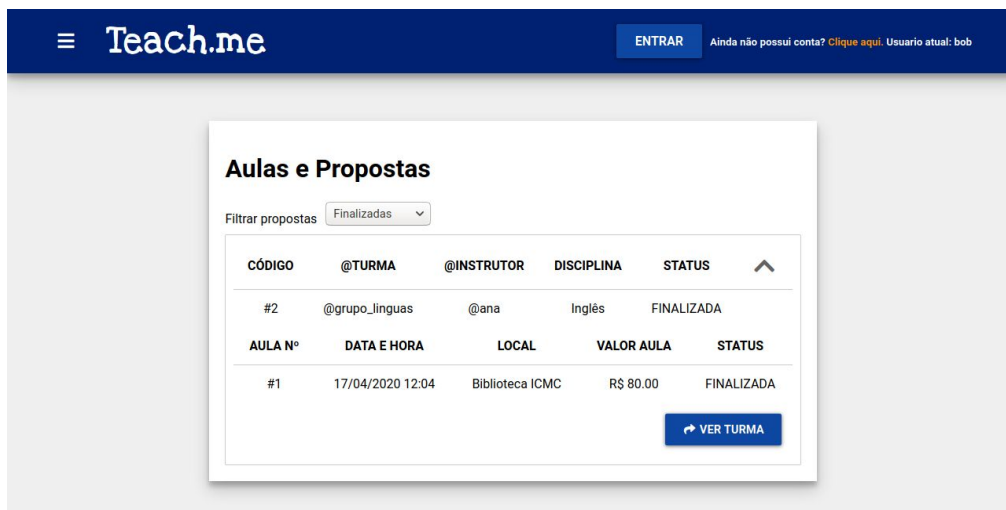


## Minhas Aulas e Propostas

**API End-point:** http://localhost/api/proposals

**Front-end:** http://localhost/painel/aulas-e-propostas

**Tela do protótipo:**



**Descrição:** Nesta tela o usuário pode visualizar as propostas em que participa, seja diretamente ou através de suas turmas. A tela também permite que as propostas sejam filtradas de acordo com seu status atual.

**Consulta Realizada:**

```
# ProposalDAO():
def select_by_student(self, student:str, status:str = ''):
    ...
    query = '''SELECT P.ID, P.TURMA, P.INSTRUTOR, P.DISCIPINA, P.CODIGO, P.STATUS,
                    TO_CHAR(P.DATA_CRIACAO:: DATE, 'dd/mm/yyyy'), P.PRECO_TOTAL,
                    A.NUMERO, A.LOCAL, A.STATUS, A.PRECO_FINAL,
                    TO_CHAR(A.DATA_INICIO:: DATE, 'dd/mm/yyyy hh:mm') ,
                    (AC.ALUNO IS NULL) ACEITO
    FROM proposta P
    INNER JOIN aula A ON (P.ID = A.PROPOSTA)
    INNER JOIN participante PA ON (P.TURMA = PA.TURMA)
    LEFT JOIN aceita AC
        ON (P.ID = AC.PROPOSTA AND PA.ALUNO = AC.ALUNO AND PA.TURMA = AC.TURMA)
    WHERE PA.ALUNO = %s AND P.STATUS = %s'''
```

## Criar Nova Turma

**API End-point:** http://localhost/api/class/register

**Front-end:** http://localhost/painel/minhas-turmas

**Tela do protótipo:**

Estudos P1 - Cálculo 2 (2020) 1/30 participantes | Situação: BUSCANDO INSTRUTOR

Turma para estudar para as provas e cálculo 2 unificado (ICMC, EESC, IFSC, IQSC) e também resolver exercícios em grupo.

### Criar nova turma

Título da turma

@Classname

Limite de participantes

10

Descrição da turma

+ CRIAR NOVA TURMA

**Descrição:** Logo ao final da tela de visualizar turmas, foi implementado também um pequeno formulário para criação de novas turmas. Após a criação, a turma é inserida no banco com o seu criador definido como Líder.

**Consulta Realizada:**

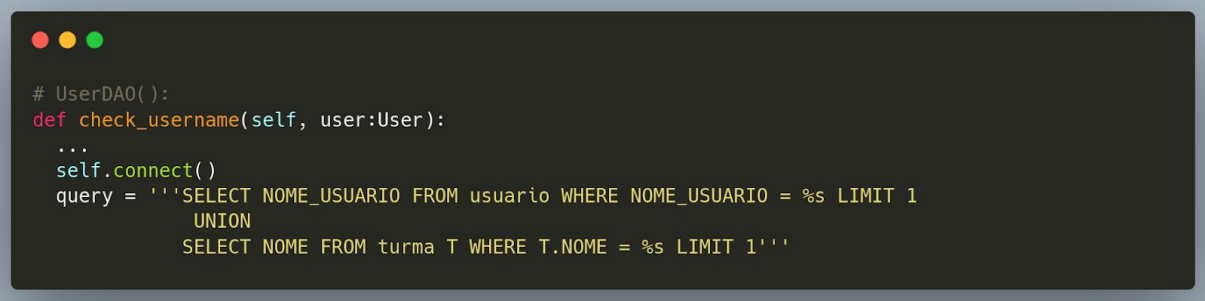
```
# ClassDAO():
def insert(self, class_:Class):
    ...
    query = '''INSERT INTO turma (NOME, TITULO, DESCRICAO, QTD_PARTICIPANTES,
                                MAX_PARTICIPANTES, SITUACAO)
              VALUES (%s, %s, %s, %s, %s, %s);'''
    ...
    self.cur.execute(query, [class_.classname, class_.title, class_.description,
                             class_.members_qtt, class_.max_members, class_.situation])
```

## Verificar Disponibilidade de Username

**API End-point:** http://localhost/api/check-username

**Descrição:** Esta funcionalidade foi criada na API para auxiliar as inserções de usuários e turmas. Ela serve para verificar se o dado 'username/classname' solicitado pelo usuário já está em uso por outro cadastro no banco de dados.

**Consulta Realizada:**



```
# UserDAO():
def check_username(self, user:User):
    ...
    self.connect()
    query = '''SELECT NOME_USUARIO FROM usuario WHERE NOME_USUARIO = %s LIMIT 1
              UNION
              SELECT NOME FROM turma T WHERE T.NOME = %s LIMIT 1'''
```

## 14. Conclusões

Para nós o projeto foi muito interessante como um todo devido, primeiramente, ao fato de o tema ser algo que já era do interesse de alguns membros do grupo. No geral, gostamos bastante do processo de acompanhamento do projeto que também nos ajudou a garantir que estamos sempre caminhando para as melhores soluções.

Um dos pontos de maior dificuldade foi o da primeira entrega, uma vez que para nós não ficou tão claro quão complexo deveria ser o sistema a ser modelado. O fato de grande parte das opções de sistemas colaborativos terem uma estrutura básica mais simples também foi um dos pontos de dificuldade e exigiu um pouco da nossa criatividade para complementar a ideia.

De aprendizado, achamos tudo muito satisfatório, pois agora conseguimos enxergar diversas melhorias que na primeira entrega não percebemos e que hoje, se tivéssemos que refazer o projeto, provavelmente sairia coisas bem mais elaboradas e interessantes.

Por fim, o único ponto de melhoria que acreditamos necessário seria diminuir um pouco a quantidade de queries médias ou ao menos tentar deixar mais definido o que elas seriam na 3ª entrega, pois sentimos um pouco de dificuldade e receio em classificar algumas de nossas queries.