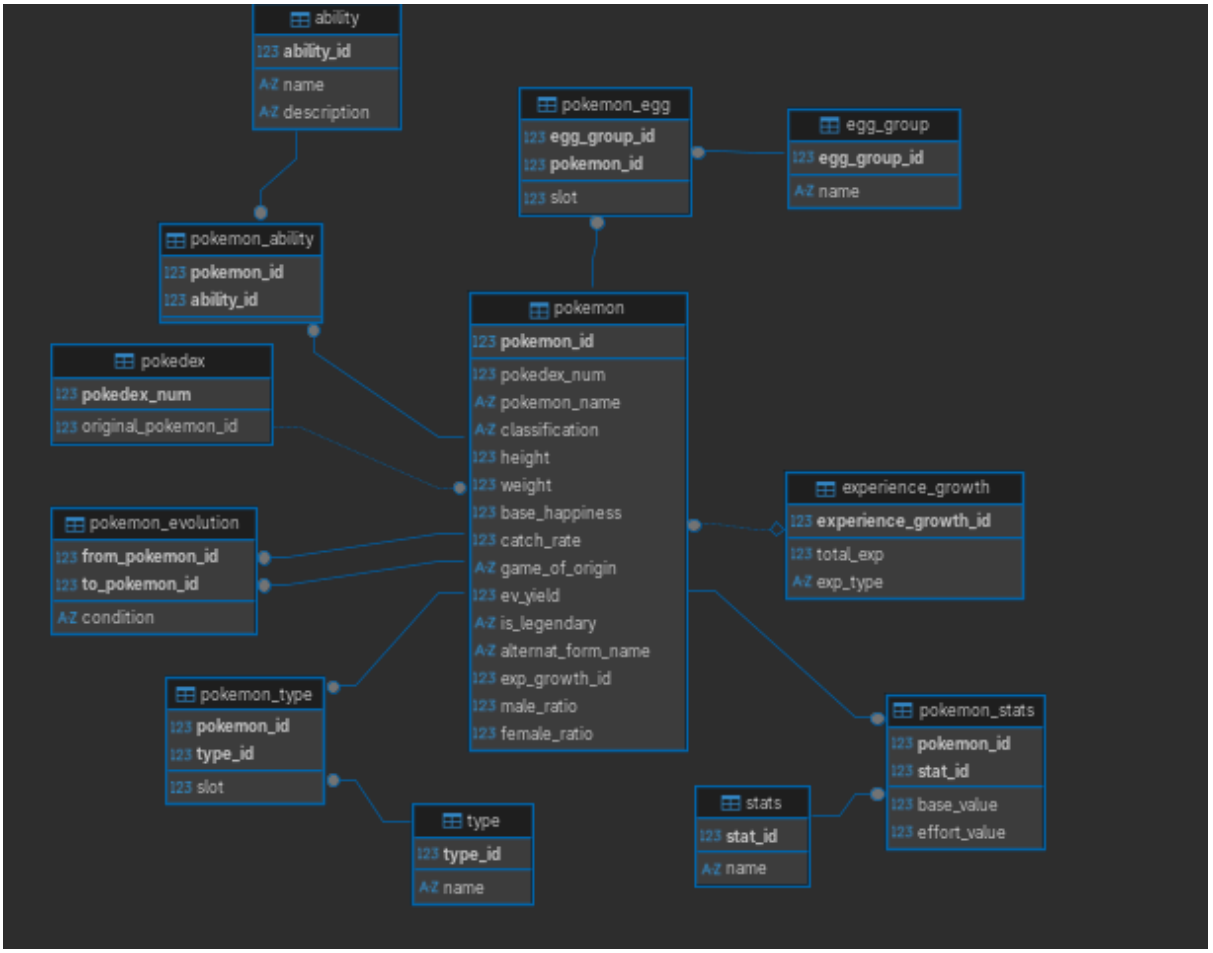
Documento de Justificativa

MER



1) Separação da tabela principal em múltiplas entidades

Problema anterior

A tabela original armazenava quase *todas as informações do Pokémon em uma única tabela*, incluindo:

- Tipos primário e secundário
- Habilidades primária/secundária/escondida
- Informações de evolução
- Grupo de ovos
- Status base
- Relações por “texto” sem FK

Isso violava:

- **1NF** (dados multivalorados em colunas)
- **3NF** (dependências repetidas)
- **Repetição massiva de strings** (tipos, habilidades, egg groups)

Justificativa da mudança

O modelo foi normalizado para:

- Evitar redundância
- Garantir integridade
- Facilitar join, consulta e indexação
- Tornar a base escalável

---

2) Criação de tabelas de relacionamento (1:N)

Novas tabelas criadas

- pokemon_type

- pokemon_ability
- pokemon_egg
- pokemon_status

Problema anterior

A tabela tinha colunas fixas como:

- **Primary Type / Secondary Type**
- **Primary Ability / Secondary Ability / Hidden Ability**
- **Primary Egg Group / Secondary Egg Group**

Essas colunas:

- Impediam 1:N natural dos dados
- Duplicavam strings
- Dificultavam extensões futuras

Justificativa

Normalizou-se para tabelas de junção que:

- Eliminar redundância
- Facilitam consultas (“quais pokémon têm Ability X?”)
- Permitem expansão sem alterar estrutura

---

3) Criação de tabelas de domínio / lookup

Criadas:

- type

- ability
- egg_group
- experience_growth
- stats

Problema anterior

Estava tudo aglomerado apenas uma tabela.

Isso causa:

- Performance lenta e complexidade de consultas
- Inconsistência de Dados
- Dificuldade de fazer manutenção

Justificativa

Tabelas lookup padronizam e permitem:

- FK garantindo integridade
- Indexação eficiente
- Zero redundância
- Otimização de JOIN

---

4) Divisão das evoluções em tabela específica (pokemon_evolution)

Problema anterior

A tabela antiga tinha:

- pre_evolution_pokemon_id

- Evolution Details

Esses campos:

- Não garantem integridade de referência
- Armazenavam condições de evolução como *texto livre*
- Não permitiam múltiplas evoluções possíveis (Eeveelutions, por exemplo)

Justificativa

A nova tabela:

- Representa *cada relação de evolução* como uma linha
- Permite múltiplas evoluções por Pokémon
- Armazena condições de forma organizada
- Mantém integridade referencial (FK para Pokémon)

---

5) Separação dos status base e EV Yield em pokemon_status

Problema anterior

No dicionário original, você tinha colunas como:

- HP Base Stat
- Attack Base Stat
- HP EV
- Attack EV
- etc.

Essas colunas:

- Violavam 1NF e 3NF (atributos repetidos)
- Tornavam consultas dinâmicas difíceis
- Impediam adição futura de novos atributos (sem mudar modelo)

Justificativa

A nova tabela pokemon_stats:

- Usa FK para tabela stats
- Permite qualquer stat definido
- Evita colunas duplicadas
- Melhora modelagem e integridade

---

6) Tabela pokedex para separar número regional e original

Problema anterior

O dicionário tinha:

- Pokémon Number
- Pokédex Original ID
- Regional Number

Esses dados misturados na entidade Pokémon causavam:

- Duplicidade (um Pokémon pode ter vários números regionais)
- Contradição no modelo (padrões diferentes por região)

Justificativa

A tabela pokedex:

- Permite múltiplos números por Pokémon
- Organiza diferença entre Pokédex Nacional vs Regionais
- Mantém integridade e flexibilidade

---

7) Tipos ajustados para formatos mais adequados

Problema anterior

Tipo inadequado gerava:

- Impossibilidade de cálculos
- Falta de validação
- Erros de importação
- Dados inconsistentes

Justificativa

Definir tipos adequados:

- Permite cálculos matemáticos
- Garante restrições
- Facilita consistência na importação de dados reais

---

8) Adição de chaves estrangeiras e primárias

No original, não havia FKs formais.

Problema anterior

- Era possível cadastrar um tipo inexistente
- Evolução apontando para Pokémon inexistente
- Habilidades inconsistentes

Justificativa

As FKs:

- Garantem integridade de dados
- Impedem registros órfãos
- Otimizam JOIN
- Melhoram a consistência da base completa

Perfeito — então vamos documentar **EXATAMENTE esses dois índices** que você criou, com:

✔ **Problema anterior**
✔ **Justificativa**
✔ **Por que o índice é necessário**
✔ **O que ele otimiza no seu modelo atual**

De forma clara e que você pode colocar no seu relatório/trabalho.

---

9) Índices

No modelo original, **nenhum índice existia além das chaves primárias**.
Isso causava:

- Lentidão para fazer JOIN entre tabelas
- Buscas demoradas em tabelas de relacionamento
- Filtros e WHERE varrendo a tabela inteira (*full table scan*)
- Desempenho ruim conforme a quantidade de Pokémon crescia

Por isso foram adicionados índices em campos frequentemente utilizados em **JOIN** e **busca direta**.

Índice 1: `idx_pokemon_pokedex_num`

CREATE INDEX idx_pokemon_pokedex_num ON pokemon(pokedex_num);

Problema anterior

No modelo antigo, consultas como:

```
SELECT *
FROM pokemon p
JOIN pokedex d ON p.pokedex_num = d.pokedex_num;
```

ou filtros como:

```
WHERE pokedex_num = 25;
```

Precisavam **varrer toda a tabela `pokémon`** para encontrar os registros correspondentes.

Isso escalava muito mal à medida que o número de Pokémon aumentava, então ficava muito pesado.

Justificativa do índice

O campo `pokedex_num` é:

- uma coluna de referência usada frequentemente em JOIN
- usada em WHERE
- usada para identificar Pokémon por número da Pokédex

Criar um índice nele faz com que o banco:

- encontre rapidamente os Pokémon pela Pokédex
- execute JOINs com `pokedex` de forma eficiente
- reduza drasticamente o custo de leitura

✔ Resultado:

Consultas mais rápidas envolvendo números da Pokédex.

Índice 2: `idx_pokemon_type_type_id`

CREATE INDEX idx_pokemon_type_type_id ON pokemon_type(type_id);

Problema anterior

A tabela `pokemon_type` é uma *tabela de ligação (N:N)* e as consultas mais comuns são:

- "Quais Pokémon são do tipo X?"
- "Quais Pokémon são do tipo Fogo?"

Exemplo:

```
SELECT p.pokemon_name
FROM pokemon_type pt
JOIN pokemon p ON p.pokemon_id = pt.pokemon_id
WHERE pt.type_id = 10;   -- ex: fogo
```

Sem índice, o banco precisaria:

Percorrer **toda a tabela pokemon_type** procurando as linhas com aquele tipo.

Com 2 tipos por Pokémon, e >1000 Pokémon, essa tabela teria >2000 registros — escalando sempre.

Justificativa

Como `type_id` é:

- muito usado em WHERE
- muito usado em JOIN
- altamente repetido (muitos Pokémon compartilham mesmo tipo)

✔ Resultado:

- Busca instantânea de Pokémon por tipo
- JOINs eficientes entre `pokemon_type` → `type`
- Redução grande no custo de consulta em análises por tipo

Justificativa dos índices:

Foram adicionados índices para acelerar:

1. **Identificação e JOIN de Pokémon pela Pokédex**
   → `idx_pokemon_pokedex_num`

2. **Busca de Pokémon por tipo**
   → `idx_pokemon_type_type_id`
