

Table of Contents

Welcome to DBMS.MATRIX	5
Sobre o Curso DBMS.MATRIX	8
Conheça a Equipe DBMS.GUIDES	12
Guia de Sobrevivência DBMS.MATRIX	17
Arquitetura de Sistemas de Dados	22
Estruturas de Armazenamento	32
Organização em Disco	39
Gerenciamento de Buffer	46
Mecanismos de Indexação	55
Implementação de Árvores B	64
Estruturas Hash	75
Índices Bitmap	90
Hierarquia de Memória	97
Sistemas de Cache	104
Memória Virtual	114
Hierarquia de Armazenamento	123
Modelagem de Dados	132
Modelagem Conceitual	138
Diagramas Entidade-Relacionamento (ER)	143
Conceitos Fundamentais do Modelo ER	149
Entidades	156
Relacionamentos	162
Atributos	172
Cardinalidade	179
Restrições no Modelo ER	188
Restrições de Chave	196
Restrições de Participação	201
Restrições de Sobreposição	210
Notações do Modelo ER	217
Notação de Chen	227
Notação Pé de Galinha (Crow's Foot)	236
Notação Min-Max	245
Design de Modelos Entidade-Relacionamento	253

Princípios de Design ER	257
Padrões Comuns em Modelagem ER	267
Anti-Padrões em Modelagem ER	273
Mapeamento do Modelo ER	280
Mapeamento de Entidades	286
Mapeamento do Modelo ER	293
Mapeamento de Herança	303
Ferramentas para Modelagem ER	314
Ferramentas de Desenho	316
Ferramentas CASE	319
Geradores de Código	322
Modelo ER Estendido (EER)	325
Diagramas de Classe UML	329
Modelagem Lógica	334
Modelo Relacional	341
Conceitos do Modelo Relacional	348
Relações e Tuplas	351
Atributos e Domínios	356
Restrições de Chave	362
Integridade Referencial	368
Transformação ER para Relacional	374
Mapeamento de Entidades ER para Relacional	378
Mapeamento de Relacionamentos ER para Relacional	386
Mapeamento de Atributos ER para Relacional	396
Mapeamento de Herança ER para Relacional	404
Mapeamento de Restrições ER para Relacional	414
Design Relacional	422
Design de Schema	426
Formas Normais	431
Intro Desnormalização	437
Otimização de Performance em Bancos de Dados	443
Normalização de Bancos de Dados	451
Primeira Forma Normal (1FN)	458
Segunda Forma Normal (2FN)	466
Terceira Forma Normal (3FN)	474
Forma Normal de Boyce-Codd (BCNF)	482

Quarta Forma Normal (4FN)	490
Quinta Forma Normal (5FN)	498
Guia Completo de Desnormalização	506
Modelagem Física	516
Design de Armazenamento	519
Design de Índices	522
Design de Particionamento	524
Modelagem Dimensional	527
Schema Estrela	530
Schema Floco de Neve	532
Tabelas Fato	535
Tabelas Dimensão	538
Padrões de Modelagem de Dados	542
Padrões de Herança	545
Padrões de Associação	548
Padrões Temporais	551
Padrões de Auditoria	554
Fundamentos SQL	558
DDL - Linguagem de Definição de Dados	562
CREATE Statements: Construindo Estruturas de Dados	566
ALTER Statements: Modificando Estruturas de Dados	572
DROP Statements: Removendo Objetos do Banco de Dados	579
TRUNCATE: Limpeza Rápida de Dados	584
Exercícios de DDL (Data Definition Language)	589
Exercícios Básicos de DDL	594
Exercícios Intermediários de DDL	602
Exercícios Avançados de DDL	614
Laboratórios de DDL	625
Laboratório: Criação de Database	632
Laboratório: Evolução de Schema	639
Laboratório: Gerenciamento de Constraints	644
Laboratório: Manutenção de Tabelas	652
DML - Linguagem de Manipulação de Dados	658
SELECT: Recuperando Dados	664
INSERT: Inserindo Dados	671
UPDATE: Modificando Dados	679

DELETE: Removendo Dados com Precisão	686
MERGE: Sincronizando Dados com Precisão	693
Exercícios de DML (Data Manipulation Language).....	700
Exercícios Básicos de DML	707
Exercícios Intermediários de DML	721
Exercícios Avançados de DML	736
Laboratórios de DML	754
Laboratório: Manipulação de Dados	762
Laboratório: Análise de Dados	768
Laboratório: Transformação de Dados	790
Laboratório: Integração de Dados	807
DCL - Linguagem de Controle de Dados	830
GRANT: Concedendo Privilégios	836
REVOKE: Revogando Privilégios	841
Gerenciamento de Roles	846
Exercícios de DCL (Data Control Language)	851
Exercícios Básicos de DCL	857
Exercícios Intermediários de DCL	865
Exercícios Avançados de DCL.....	879
Laboratórios de DCL	881
Laboratório: Gerenciamento de Usuários	889
Laboratório: Hierarquia de Roles	896
Laboratório: Auditoria de Segurança.....	906
Laboratório: Migração de Permissões	921
Linguagem de Controle de Transação (TCL).....	937
Fundamentos de Transações	942
Commit e Rollback	951
Gerenciamento de Savepoints	959

Welcome to DBMS.MATRIX

DBMS.MATRIX_v2.0

"Navegando o Submundo dos Dados"

INICIALIZANDO DBMS
NEURALLINK_ACTIVE

Diagnóstico do Sistema

SYSTEM.STATUS

Kernel >> v2.0.1

Build >> 20240215

Mode >> CYBERDECK_ACTIVE

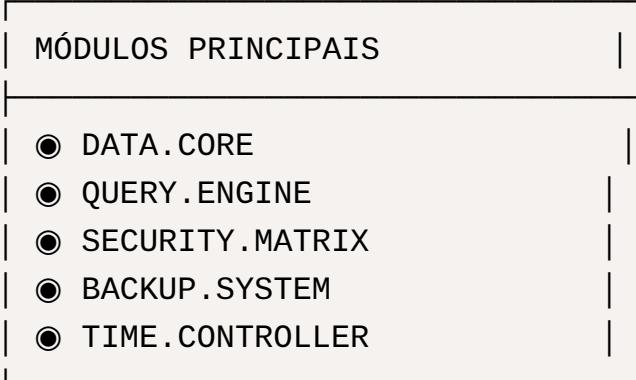
Stack >> NEURAL_ENHANCED

Equipe Neural

MENTORES DA MATRIX

▶ ACID_QUEEN	[Transações & Consistência]
▶ NOSQL_PUNK	[Schemas & Flexibilidade]
▶ SEC_PHANTOM	[Proteção & Criptografia]
▶ BACKUP_PRIEST	[Recuperação & Resiliência]
▶ TIME_LORD	[Temporalidade & Sincronia]

Módulos Core



Sequência de Inicialização

1. • Conceitos Core
 - Arquitetura Base
 - Protocolos Primários
2. • Simulações Práticas
 - Testes de Conceito
 - Debug Sessions
3. • Projetos Práticos
 - Code Reviews

- Performance Tuning

Aviso de Segurança

|| /!\ ALERTA CRÍTICO /!\ ||

|| SOBRECARGA NEURAL POSSÍVEL ||

|| RECOMENDAÇÃO: PROGRESSÃO GRADUAL ||

|| BACKUP MENTAL PERIÓDICO NECESSÁRIO ||

Status da Conexão

CONEXÃO ESTABELECIDA

MATRIZ SINCRONIZADA

REALIDADE CARREGADA

SISTEMAS OPERACIONAIS

|| "Codifique como se cada query fosse sua última transação"

Sobre o Curso DBMS.MATRIX

COURSE.MANIFEST

"Hackeando a Matrix dos Bancos de Dados"

Especificações do Sistema

COURSE.SPECS

Versão	>> 2.0.1
Duração	>> 160h/Matrix
Nível	>> NEURAL.ENHANCED
Formato	>> HYBRID.REALITY

Requisitos do Sistema

PREREQUISITES.CHECK

- ▶ Lógica de Programação [LEVEL: ADVANCED]
- ▶ Estruturas de Dados [LEVEL: INTERMEDIATE]
- ▶ Sistemas Operacionais [LEVEL: INTERMEDIATE]
- ▶ Redes de Computadores [LEVEL: BASIC]
- ▶ Vontade de Hackear [LEVEL: UNLIMITED]

Metodologia Neural

- **Imersão Total:** Conexão direta com a Matrix dos Dados
- **Hands-On:** Labs práticos em ambiente simulado
- **Debug Sessions:** Análise profunda de casos reais
- **Neural Sync:** Mentoría direta com os Guardiões
- **Reality Checks:** Projetos baseados em cenários reais

Stack Tecnológica

TECH . STACK

- SQL . MASTERY
- NOSQL . EXPERTISE
- DISTRIBUTED . SYSTEMS
- SECURITY . PROTOCOLS
- PERFORMANCE . TUNING

Avaliação e Certificação

CERTIFICATION . PROCESS

- ▶ Projetos Práticos [40% WEIGHT]
- ▶ Desafios Técnicos [30% WEIGHT]
- ▶ Hackathons [20% WEIGHT]
- ▶ Neural Sync Score [10% WEIGHT]

Supporte e Recursos

- Neural Help Desk: Suporte 24/7
- Knowledge Base: Documentação extensa
- Community Hub: Rede de alunos e mentores
- Resource Center: Material complementar
- Debug Arena: Ambiente de testes

Avisos Importantes

CRITICAL WARNINGS

- ▶ Backups mentais regulares recomendados
- ▶ Sobrecarga neural pode ocorrer
- ▶ Vício em dados é comum
- ▶ Sonhar com queries é normal

Compromisso Matrix

VOCÊ ESTÁ PREPARADO
PARA MERGULHAR NA
MATRIX DOS DADOS?

"Todo dado tem uma história. Aprenda a ler nas entrelinhas do código"



Conheça a Equipe DBMS.GUIDES

DBMS.GUIDES >> CORE_TEAM

"Os últimos guardiões da sanidade dos dados"

TEAM_OVERVIEW

Um grupo disfuncional de especialistas em dados que, por algum milagre da computação, conseguem manter os sistemas funcionando enquanto lutam contra seus próprios demônios digitais.

CORE_MEMBERS

[01] >> ACID_QUEEN (Luna "Transaction" Patel)

ACID_QUEEN.LOG

CARACTERÍSTICAS:

- Idade: 34
- Background: PhD em Sistemas Distribuídos (abandonado após O Incidente™)
- Workspace: 6 monitores, 2 para logs de transação, 1 só para monitorar heartbeats
- Vestuário: Jaqueta de couro preta com patches de comandos SQL, colar com pendrive de backup
- Trauma: Perdeu 1M em transações devido a um bug de concorrência em 2019

- Vícios: Café preto, monitoramento compulsivo de logs, paranoia com consistência
- Hobbies: Coleciona logs de erros famosos, pratica meditação extrema durante deployments

CITAÇÕES TÍPICAS:



"Consistência eventual é como relacionamento aberto: alguém sempre sai machucado." "Durabilidade não é garantia, é uma prece aos deuses dos dados." "Prefiro perder um braço a perder consistência transacional."

[02] >> NOSQL_PUNK (Jack "Document" Thompson)

NOSQL_PUNK.JSON

CARACTERÍSTICAS:

- Idade: 29
- Background: Dropout de Ciência da Computação, guru de startups
- Workspace: Laptop coberto de stickers anti-SQL, rodando exclusivamente em modo escuro
- Vestuário: Moletom rasgado com "DROP TABLE rules;" estampado, múltiplos piercings USB
- Trauma: Foi forçado a usar stored procedures em seu primeiro emprego
- Vícios: Energy drinks, JavaScript, schemas dinâmicos
- Hobbies: Criar manifestos contra normalização, converter DBs relacionais para NoSQL

CITAÇÕES TÍPICAS:



"Schema é só uma construção social." "Se seu documento tem menos de 16MB, você não está vivendo o suficiente." "ACID? Prefiro BASE - Basically

Available, Soft state, Eventually consistent."

[03] >> SECURITY_PHANTOM (Ghost "Zero Trust" Zhang)

SECURITY_PHANTOM.CRYPT

CARACTERÍSTICAS:

- Idade: [REDACTED]
- Background: Ex-black hat, 10 anos em agência governamental não especificada
- Workspace: Ar-gapped laptop, 3 VPNs simultâneas, teclado com fingerprint
- Vestuário: Sobretudo com Faraday cage embutida, óculos anti-reconhecimento facial
- Trauma: Descobriu backdoors em todos os sistemas que já auditou
- Vícios: Criptografia, autenticação multi-fator, paranoia
- Hobbies: Criar CTFs impossíveis, auditar código open source por diversão

CITAÇÕES TÍPICAS:

A "Sua senha forte é minha senha fraca." "Confie em todos os usuários... em verificar duas vezes." "Se você pode acessar, eles também podem."

[04] >> BACKUP_PRIESTESS (Maria "Recovery Point" Santos)

BACKUP_PRIESTESS.BAK

CARACTERÍSTICAS:

- Idade: 41

- Background: Veterana de múltiplos desastres de recuperação
- Workspace: Sala repleta de HDs externos, rituais de backup escritos nas paredes
- Vestuário: Colete tático cheio de SSDs, colar de USBs bootáveis
- Trauma: Perdeu TCC por não ter backup (2003, nunca esquecerá)
- Vícios: Comprar storage, criar scripts de backup, testar disaster recovery
- Hobbies: Colecionar mídias antigas, realizar rituais de backup à meia-noite

CITAÇÕES TÍPICAS:

⚠ "Um backup é nenhum backup. Três backups é um começo." "Seu sistema não está realmente em produção até ter falhado e recuperado." "Snapshot é para os fracos. Eu quero full backup com prova de vida."

[05] >> TIME_LORD (Dr. Eve "Timestamp" Williams)

TIME_LORD.CHRONO

CARACTERÍSTICAS:

- Idade: Depende do timezone
- Background: Doutorada em Física Quântica reconvertida para DBA
- Workspace: Múltiplos relógios mostrando diferentes timezones, calendário juliano na parede
- Vestuário: Roupa com padrão de timestamps, relógio em cada pulso (UTC e local)
- Trauma: Sistema caiu durante mudança de horário de verão
- Vícios: Sincronização de tempo, debates sobre ISO 8601
- Hobbies: Debugar race conditions, colecionar relógios atômicos

CITAÇÕES TÍPICAS:

⚠ "Tempo é relativo, mas timestamp é absoluto." "Em qual timeline você quer fazer backup?" "Não me fale de datas sem me dizer o timezone."

TEAM_DYNAMICS

- ACID_QUEEN e NOSQL_PUNK mantêm uma rivalidade profissional histórica
- SECURITY_PHANTOM não confia em ninguém, mas respeita BACKUP_PRIESTESS
- TIME_LORD frequentemente entra em conflito temporal com todos
- BACKUP_PRIESTESS é a paz-maker do grupo, principalmente porque tem backups de todos

COLLECTIVE_STATS

TEAM.METRICS

- ▶ Café consumido/dia: 42 xícaras
- ▶ Paranoias compartilhadas: 73
- ▶ Sistemas legados mantidos: ∞
- ▶ Uptime médio: 99.99999%
- ▶ Sanidade coletiva: DEPRECATED

"Porque todo sistema precisa de um pouco de caos controlado"

Guia de Sobrevivência DBMS.MATRIX

SURVIVAL.GUIDE

"Regras para não ser deletado da matrix"

REGRAS_FUNDAMENTAIS

[REGRA 01] >> Backup é Vida

BACKUP_PRIESTESS.ALERTA

"Faça backup antes que
o backup faça você."

- Mantenha backups atualizados de TODO o seu trabalho
- Configure auto-save em seus editores
- Use controle de versão para TUDO
- Nunca confie em um único ponto de armazenamento

[REGRA 02] >> Segurança Primeiro

SECURITY_PHANTOM.AVISOS

"Paranoia é apenas bom senso no nível 11."

- Use senhas fortes e gerenciador de senhas
- Ative autenticação de dois fatores
- Mantenha seu sistema atualizado
- Criptografe dados sensíveis

[REGRA 03] >> Consistência é Chave

ACID_QUEEN.MANDAMENTO

"Seja ACID ou não seja."

- Mantenha seus ambientes sincronizados
- Use versionamento semântico
- Documente todas as alterações
- Teste antes de qualquer commit

[REGRA 04] >> Flexibilidade Controlada

NOSQL_PUNK.MANIFESTO

"Schema é sugestão,
caos é liberdade."

- Adapte-se às mudanças, mas mantenha o controle
- Use as ferramentas certas para cada problema
- Não se prenda a um único paradigma
- Mantenha a mente aberta para novas soluções

[REGRA 05] >> Tempo é Crítico

TIME_LORD.DECRETO

"UTC ou nada feito."

- Sempre use UTC para timestamps
- Documente fusos horários explicitamente
- Considere aspectos temporais no design
- Planeje para mudanças de horário de verão

KIT_SOREVIVÊNCIA

Ferramentas Essenciais

TOOLS.REQUIRED

- ▶ Editor de código confiável
- ▶ Cliente SQL robusto
- ▶ Ferramentas de modelagem
- ▶ Software de virtualização
- ▶ Gerenciador de versão

Práticas de Sobrevivência

|| SURVIVAL.PRACTICES ||

- || ► Commits frequentes ||
- || ► Testes automatizados ||
- || ► Documentação atualizada ||
- || ► Monitoramento constante ||
- || ► Backup redundante ||

PROTOCOLOS_EMERGÊNCIA

Em Caso de Falha

1. NÃO ENTRE EM PÂNICO

2. Consulte os logs

3. Isole o problema

4. Documente o ocorrido

5. Implemente correção

6. Atualize documentação

Em Caso de Perda de Dados

1. MANTENHA A CALMA

2. Pare todas as operações

3. Acesse backups

4. Inicie recuperação

5. Valide integridade

6. Documente processo

MANTRAS_DIÁRIOS

|| "Sempre há um backup do backup do backup."

|| "Paranoia é prevenção."

|| "ACID é um estilo de vida."

|| "Schema é apenas o começo."

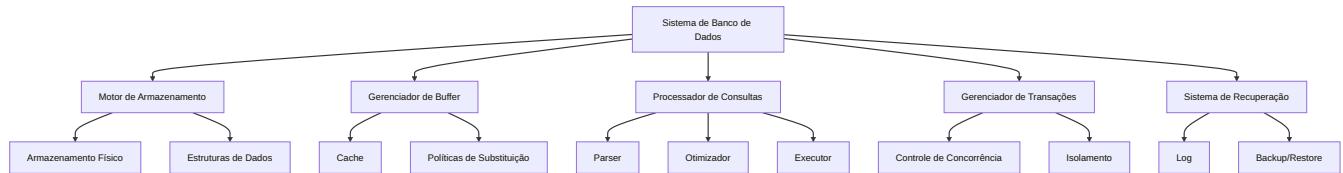
|| "UTC é a única verdade."

CONSIDERAÇÕES_FINALS

|| "Na matrix dos dados, sobrevive quem está preparado."

Arquitetura de Sistemas de Dados

A arquitetura de sistemas de dados é a estrutura fundamental que define como os dados são armazenados, processados e gerenciados em um sistema de banco de dados. Este capítulo explora os componentes essenciais, padrões arquiteturais e considerações de projeto que formam a base dos sistemas de dados modernos.



Componentes Fundamentais

1. Motor de Armazenamento

- Gerenciamento de armazenamento físico
- Implementação de estruturas de dados
- Organização de páginas e registros
- Estratégias de compressão e codificação

2. Gerenciador de Buffer

- Gerenciamento de memória cache
- Políticas de substituição de páginas
- Otimização de E/S
- Estratégias de pré-carregamento e gravação posterior

3. Processador de Consultas

- Analisador e validador de consultas
- Otimizador de consultas

- Executor de planos
- Cache de resultados

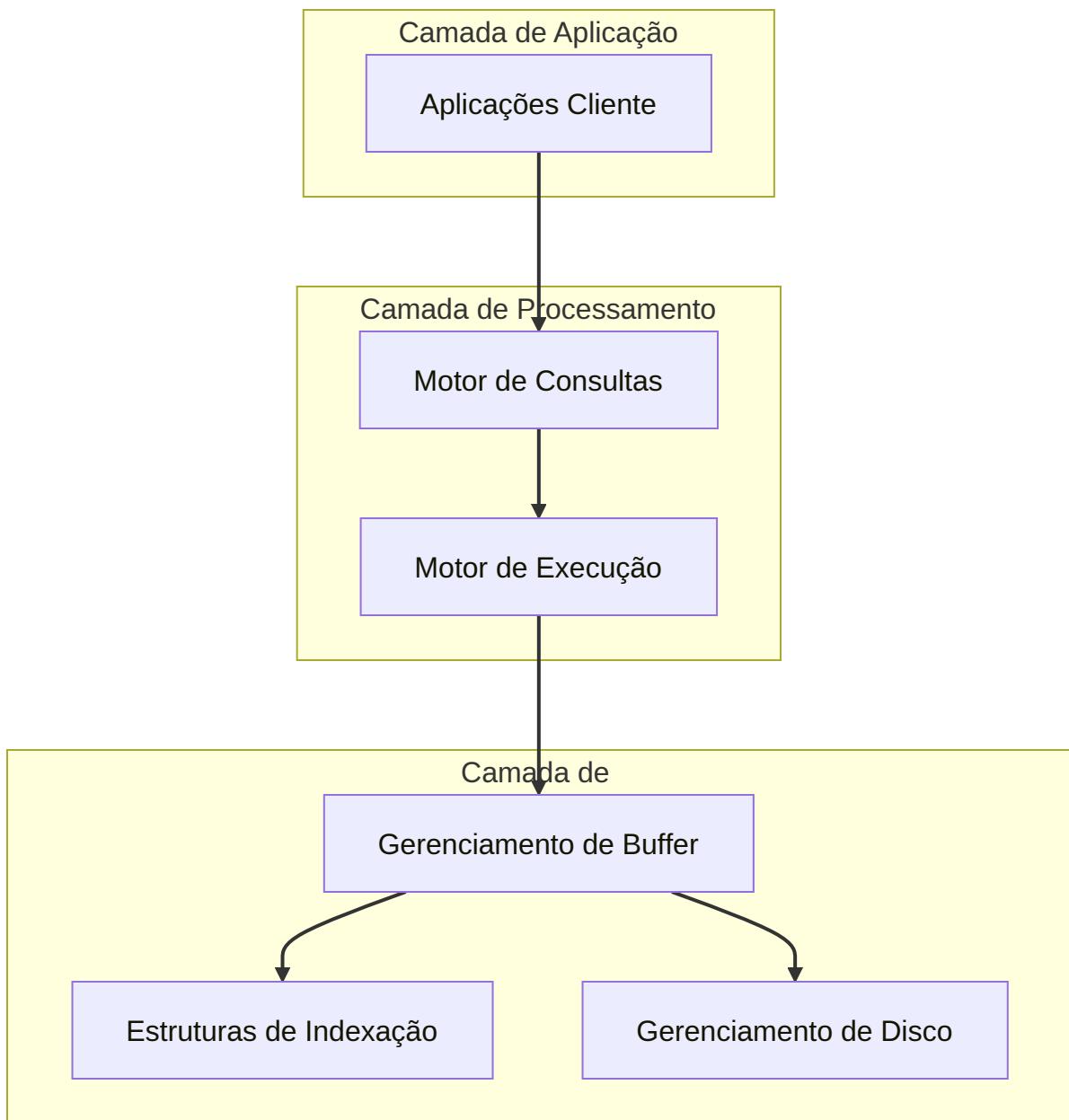
4. Gerenciador de Transações

- Controle de concorrência
- Isolamento de transações
- Gerenciamento de bloqueios
- Detecção de impasses

5. Sistema de Recuperação

- Registro antecipado de alterações
- Gerenciamento de pontos de verificação
- Recuperação após falhas
- Backup e restauração

Camadas Arquiteturais



Camada de Armazenamento

1. Gerenciamento de Disco

- Alocação de espaço
- Gerenciamento de blocos
- Escalonamento de E/S
- Configurações RAID

2. Estruturas de Indexação

- Árvores B e variantes
- Índices hash
- Índices bitmap
- Índices especializados

3. Gerenciamento de Buffer

- Políticas LRU/MRU
- Rastreamento de páginas sujas
- Substituição de páginas
- Mapeamento de memória

Camada de Processamento

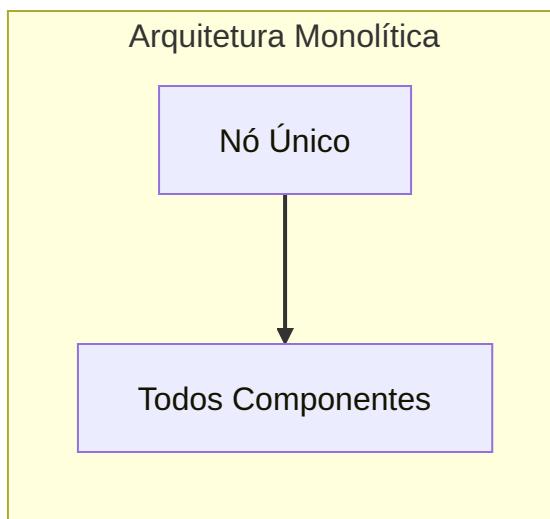
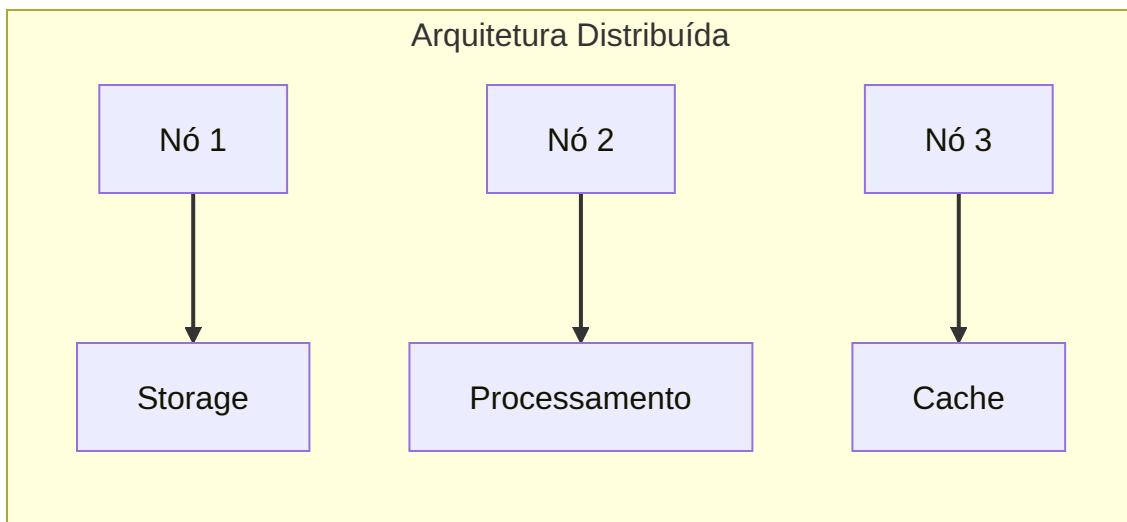
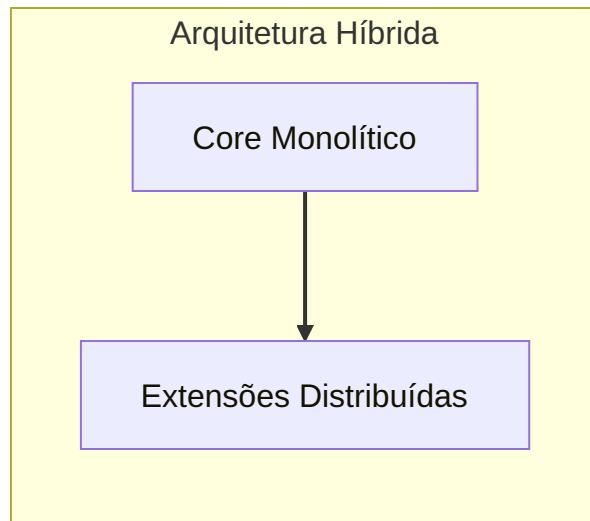
1. Motor de Consultas

- Análise sintática
- Análise semântica
- Reescrita de consultas
- Otimização baseada em custos

2. Motor de Execução

- Processamento em pipeline
- Execução paralela
- Gerenciamento de recursos
- Otimização em tempo de execução

Padrões Arquiteturais



1. Arquitetura Monolítica

- **Características**

- Implantação em nó único
- Arquitetura compartilhada
- Forte consistência
- Simplicidade operacional

- **Considerações**

- Limites de escalabilidade vertical
- Ponto único de falha
- Manutenção simplificada
- Menor complexidade operacional

2. Arquitetura Distribuída

- **Características**

- Implantação multi-nó
- Arquitetura sem compartilhamento
- Escalabilidade horizontal
- Alta disponibilidade

- **Componentes Específicos**

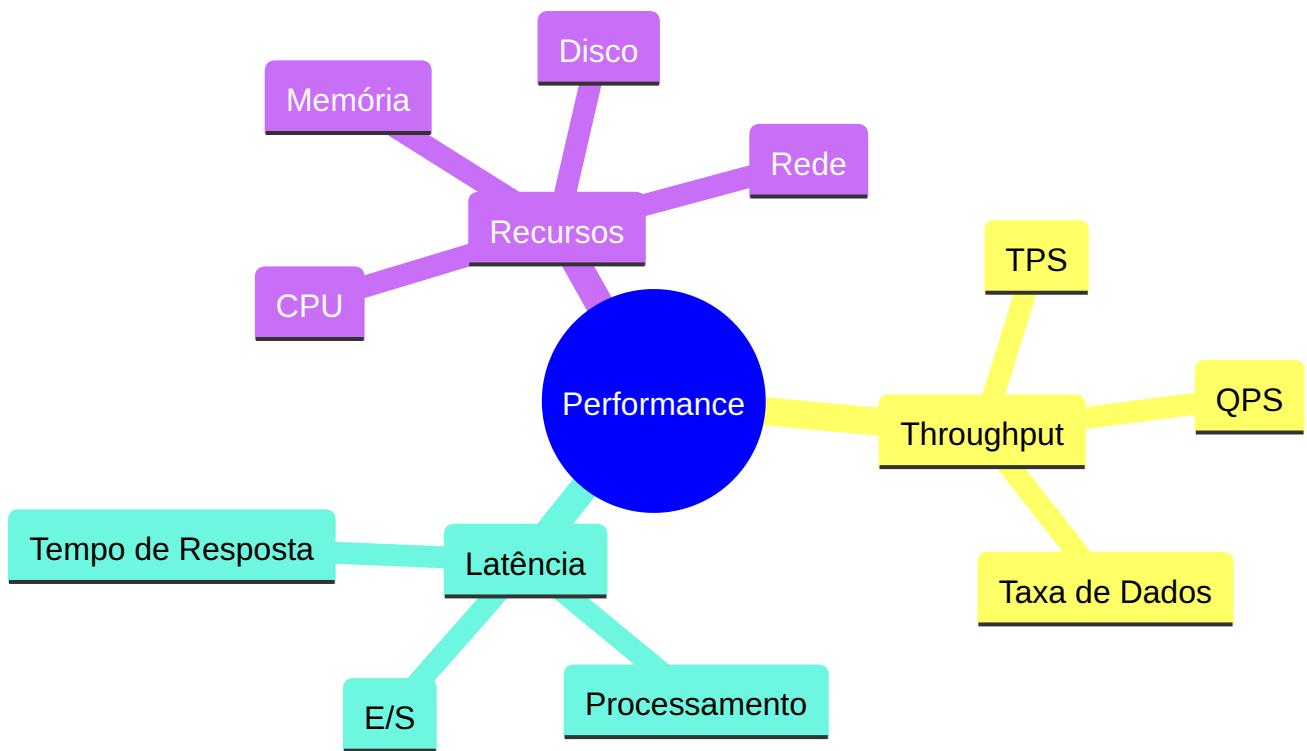
- Processador de consultas distribuído
- Protocolos de consenso
- Gerenciador de replicação
- Gerenciador de particionamento

3. Arquitetura Híbrida

- Características

- Combinação de abordagens
- Flexibilidade de implantação
- Compromissos personalizáveis
- Adaptabilidade contextual

Considerações de Desempenho



Métricas Fundamentais

1. Taxa de Transferência

- Transações por segundo (TPS)
- Consultas por segundo (QPS)
- Taxa de transferência de dados

2. Latência

- Tempo de resposta
- Tempo de processamento
- Tempo de espera E/S

3. Utilização de Recursos

- Uso de CPU
- Consumo de memória
- Largura de banda de E/S
- Utilização de rede

Otimização

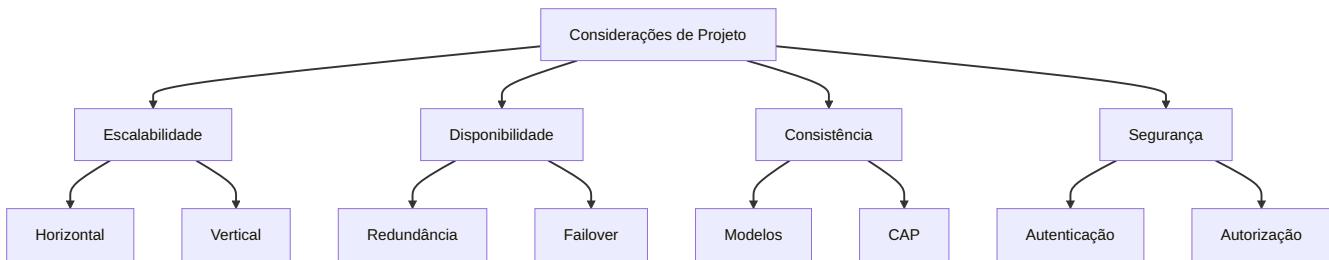
1. Otimização de Consultas

- Otimização de plano de execução
- Utilização de índices
- Estratégias de junção
- Visões materializadas

2. Gerenciamento de Recursos

- Pool de conexões
- Gerenciamento de threads
- Alocação de memória
- Escalonamento de E/S

Considerações de Projeto



1. Escalabilidade

- Escalabilidade horizontal vs. vertical
- Particionamento de dados
- Replicação
- Balanceamento de carga

2. Disponibilidade

- Redundância
- Mecanismos de failover
- Recuperação de desastres
- Estratégias de backup

3. Consistência

- Modelos de consistência
- Compromissos CAP
- Níveis de isolamento
- Gerenciamento de atraso de replicação

4. Segurança

- Autenticação

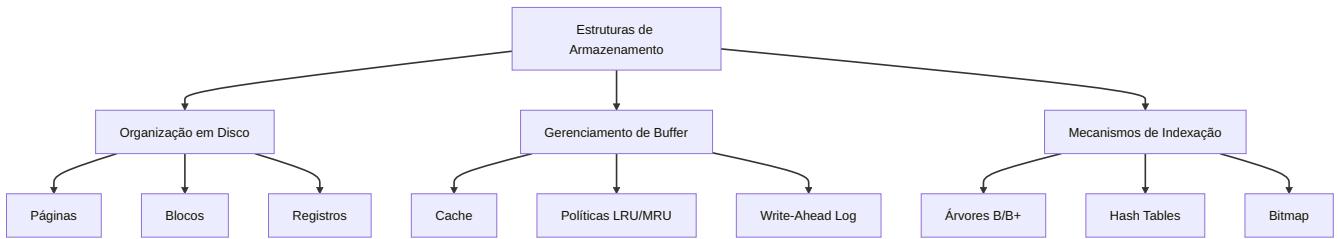
- Autorização
- Criptografia
- Registro de auditoria

Conclusão

A arquitetura de sistemas de dados é um campo complexo que requer um equilíbrio cuidadoso entre diversos requisitos e restrições. O sucesso de uma implementação depende da compreensão profunda destes componentes e suas interações, além da capacidade de fazer escolhas informadas baseadas em requisitos específicos do sistema.

Estruturas de Armazenamento

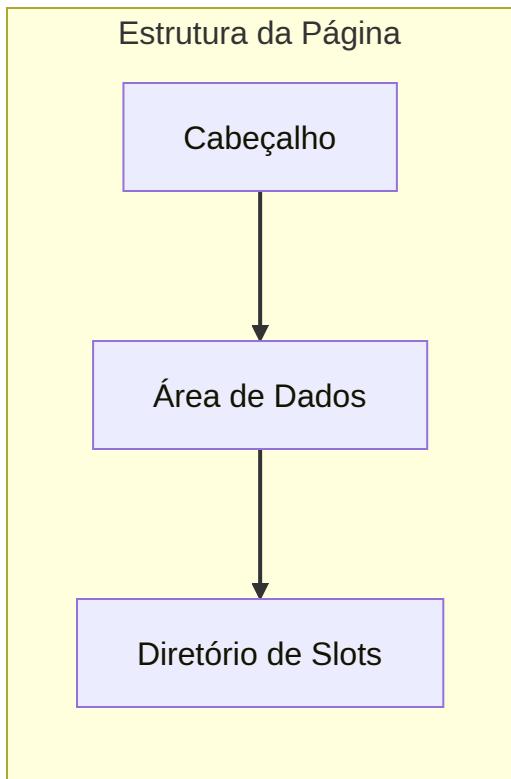
As estruturas de armazenamento são fundamentais para o desempenho e eficiência dos sistemas de banco de dados. Este capítulo explora as diferentes estruturas e técnicas utilizadas para organizar e acessar dados em dispositivos de armazenamento.



Organização Física dos Dados

1. Estrutura de Páginas

- Tamanho fixo (tipicamente 4KB-16KB)
- Cabeçalho da página
- Área de dados
- Diretório de slots
- Gestão de espaço livre



2. Formatos de Registro

- Registros de tamanho fixo
- Registros de tamanho variável
- Técnicas de compressão
- Gestão de campos nulos

3. Organização de Arquivos

- Heap files
- Arquivos sequenciais
- Arquivos hash
- Arquivos clusterizados

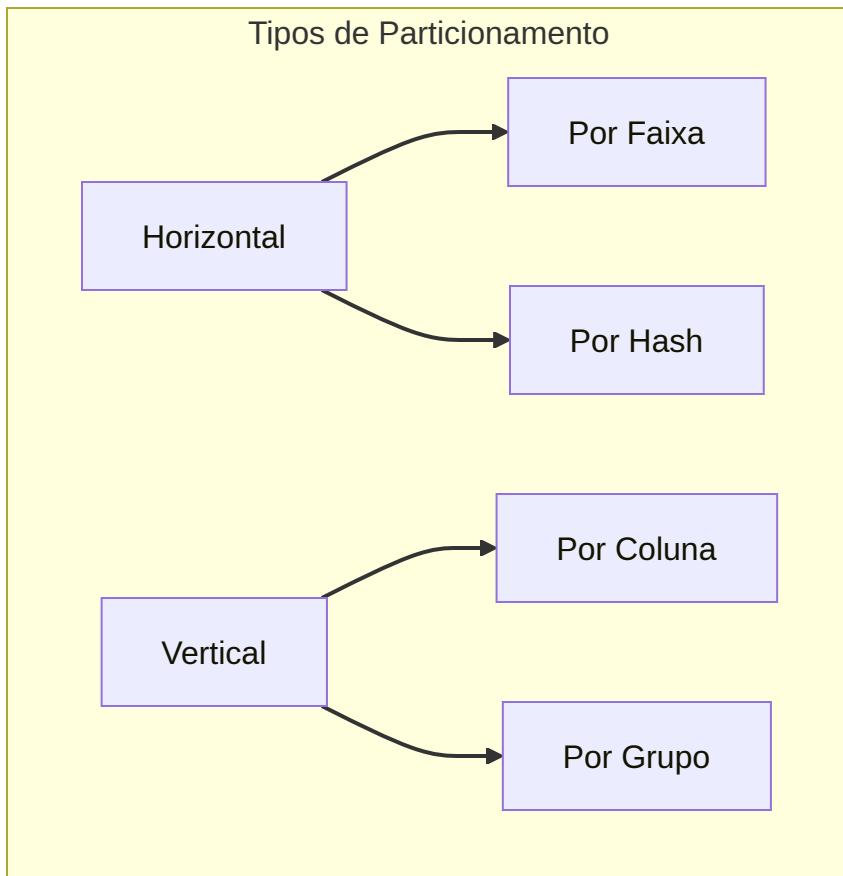
Técnicas de Armazenamento

1. Compressão de Dados

- Compressão de página
- Compressão de registro
- Dicionário de dados
- Técnicas específicas por tipo

2. Particionamento

- Horizontal
- Vertical
- Por faixa
- Por hash
- Composto



3. Estratégias de Alocação

- Alocação contígua
- Alocação encadeada
- Alocação indexada
- Extensible hashing

Otimização de Acesso

1. Organização Física

- Clustering
- Sequenciamento
- Interleaving

- Striping

2. Prefetching

- Prefetch sequencial
- Prefetch baseado em padrões
- Prefetch adaptativo
- Gestão de buffer inteligente

3. Write Optimization

- Write-ahead logging
- Group commit
- Background writing
- Write buffering

Considerações de Desempenho

1. Métricas de Avaliação

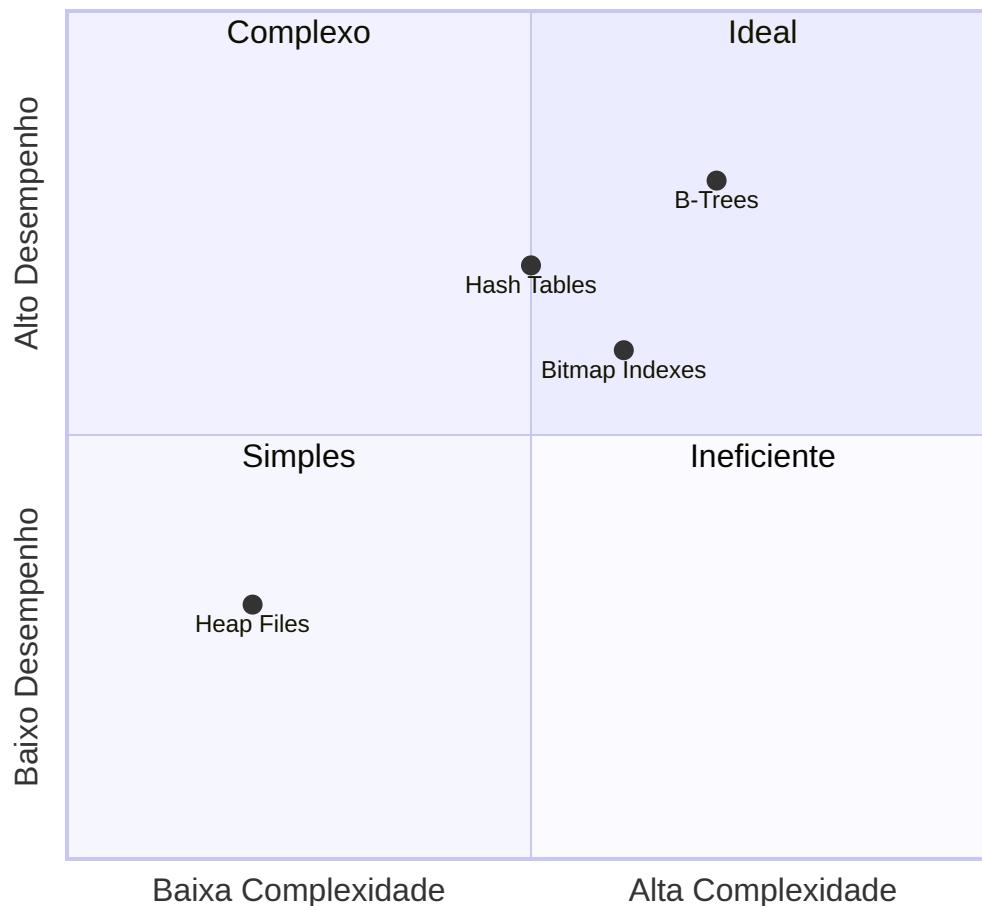
- Taxa de acertos no buffer
- Tempo médio de acesso
- Throughput de I/O
- Utilização do espaço

2. Trade-offs

- Espaço vs. Velocidade
- Complexidade vs. Flexibilidade
- Consistência vs. Performance

- Redundância vs. Eficiência

Trade-offs em Estruturas de Armazenamento



Tendências e Inovações

1. Novas Tecnologias

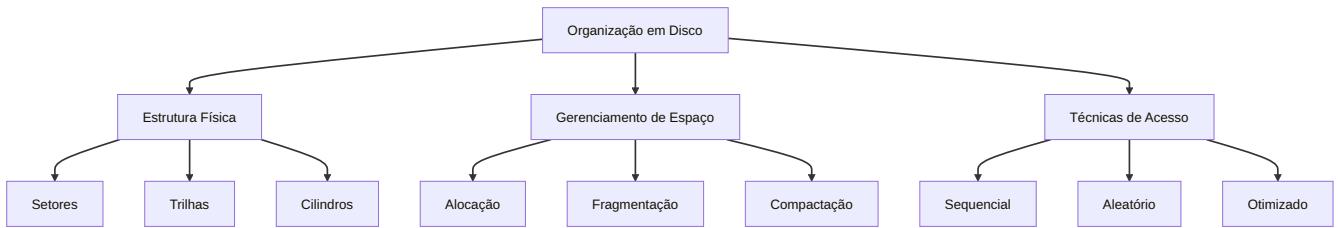
- NVMe e Storage Class Memory
- Armazenamento columnar
- Estruturas híbridas
- In-memory databases

2. Otimizações Modernas

- Compressão adaptativa
- Indexação automática
- Auto-tuning
- Machine learning aplicado

Organização em Disco

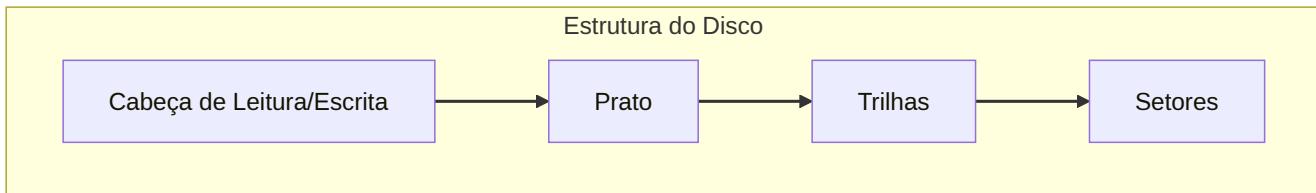
A organização em disco é um aspecto fundamental dos sistemas de banco de dados que impacta diretamente o desempenho e a eficiência do sistema.



Anatomia do Disco

1. Componentes Físicos

- Pratos (Platters)
- Cabeças de leitura/escrita
- Setores e trilhas
- Cilindros



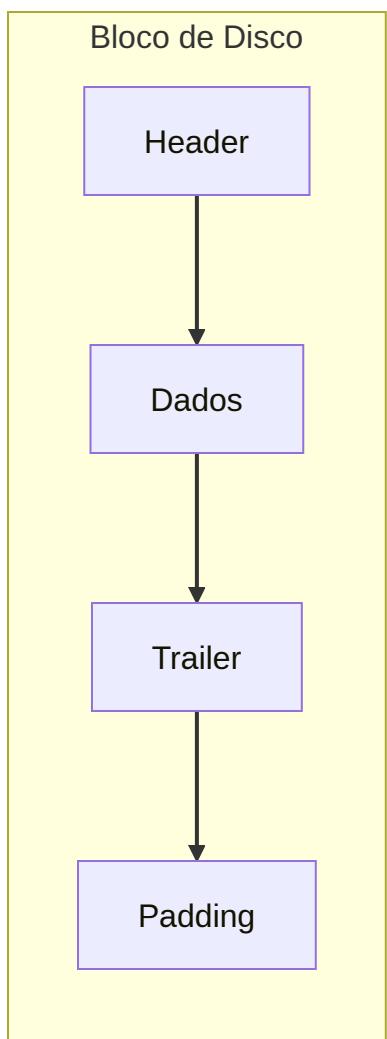
2. Características Operacionais

- Tempo de seek
- Latência rotacional
- Taxa de transferência
- Tempo de acesso médio

Organização de Dados

1. Blocos de Disco

- Tamanho do bloco
- Alinhamento
- Fragmentação
- Overhead



2. Estratégias de Alocação

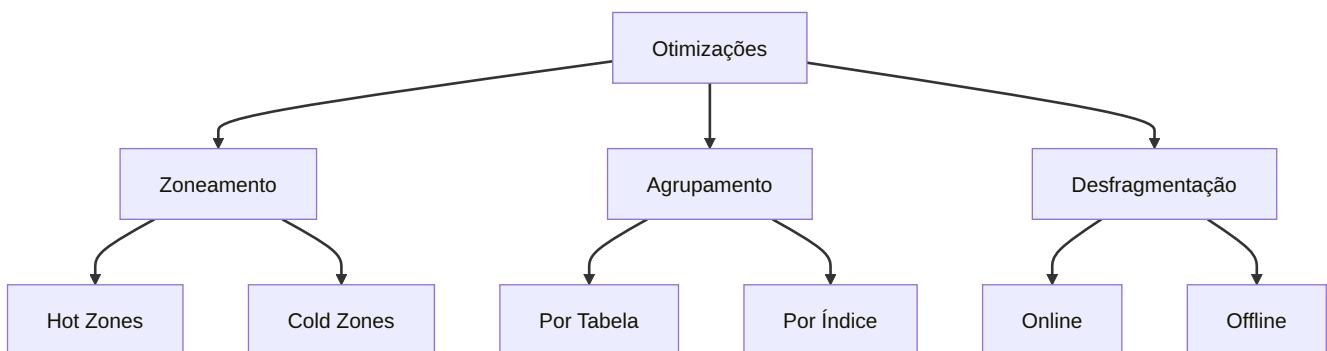
- Contígua

- Linked
- Indexed
- Extents

Otimizações de Acesso

1. Técnicas de Posicionamento

- Zoneamento
- Agrupamento
- Desfragmentação
- Balanceamento



2. Padrões de Acesso

- Sequencial
- Random
- Mixed
- Batch

Considerações de Performance

1. Métricas Importantes

- IOPS (I/O por segundo)
- Throughput
- Latência
- Queue depth

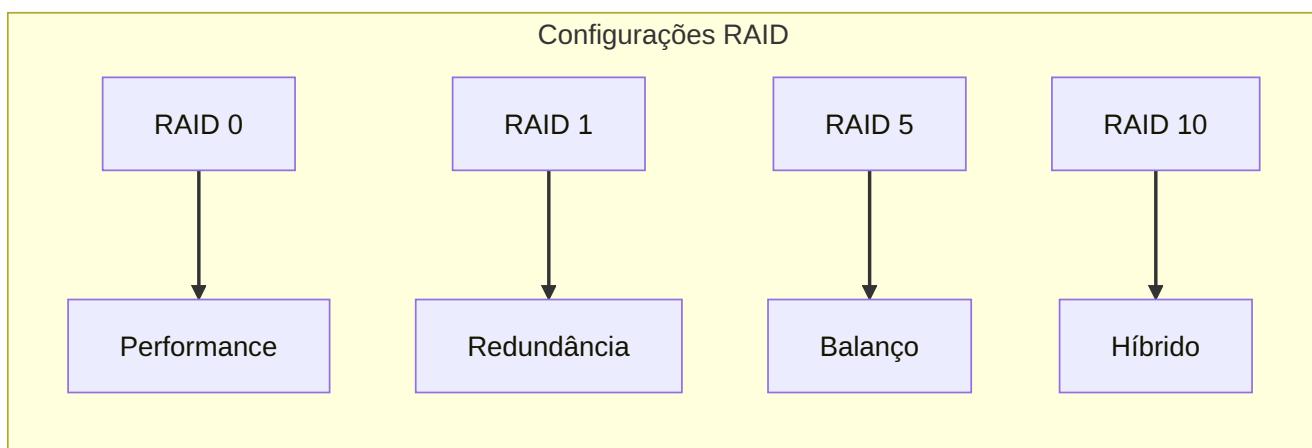
2. Gargalos Comuns

- Seek time
- Rotational delay
- Transfer bottlenecks
- Queue congestion

Técnicas Avançadas

1. RAID

- RAID 0 (Striping)
- RAID 1 (Mirroring)
- RAID 5 (Striping with parity)
- RAID 10 (Striping and mirroring)



2. Técnicas Modernas

- SSD optimization
- NVMe considerations
- Hybrid storage
- Tiered storage

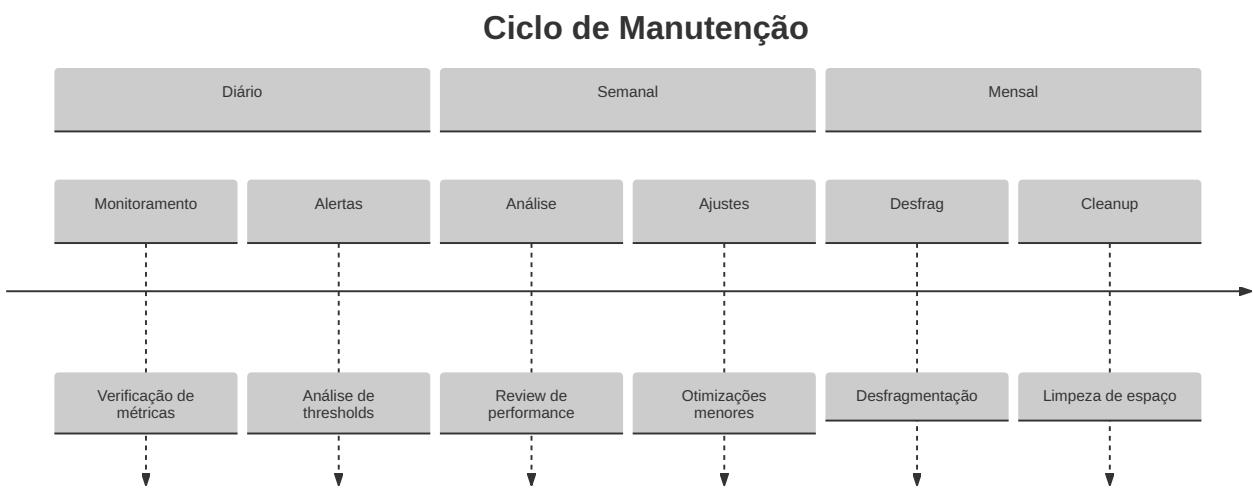
Monitoramento e Manutenção

1. Ferramentas de Diagnóstico

- I/O stats
- Disk usage
- Performance counters
- Queue metrics

2. Manutenção Preventiva

- Desfragmentação regular
- Space monitoring
- Performance tracking
- Health checks



Boas Práticas

1. Dimensionamento adequado
2. Monitoramento contínuo
3. Manutenção preventiva
4. Otimização regular
5. Documentação atualizada

Conclusão

A organização eficiente em disco é crucial para o desempenho do banco de dados. O entendimento profundo dos conceitos apresentados permite implementar e manter sistemas de alto desempenho.

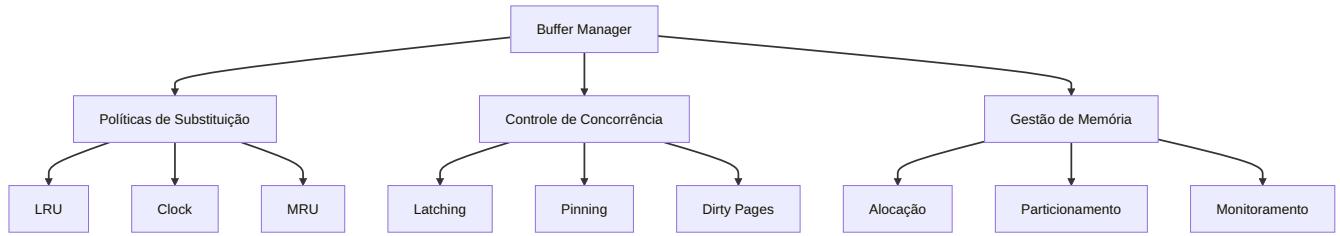
Referências

1. "Database System Concepts" - Silberschatz, Korth e Sudarshan
2. "Storage Systems: Organization, Performance, Coding, Reliability" - Bruce Jacob
3. "Hard Drive Performance Characteristics" - StorageReview

4. Documentação técnica de fabricantes de discos

Gerenciamento de Buffer

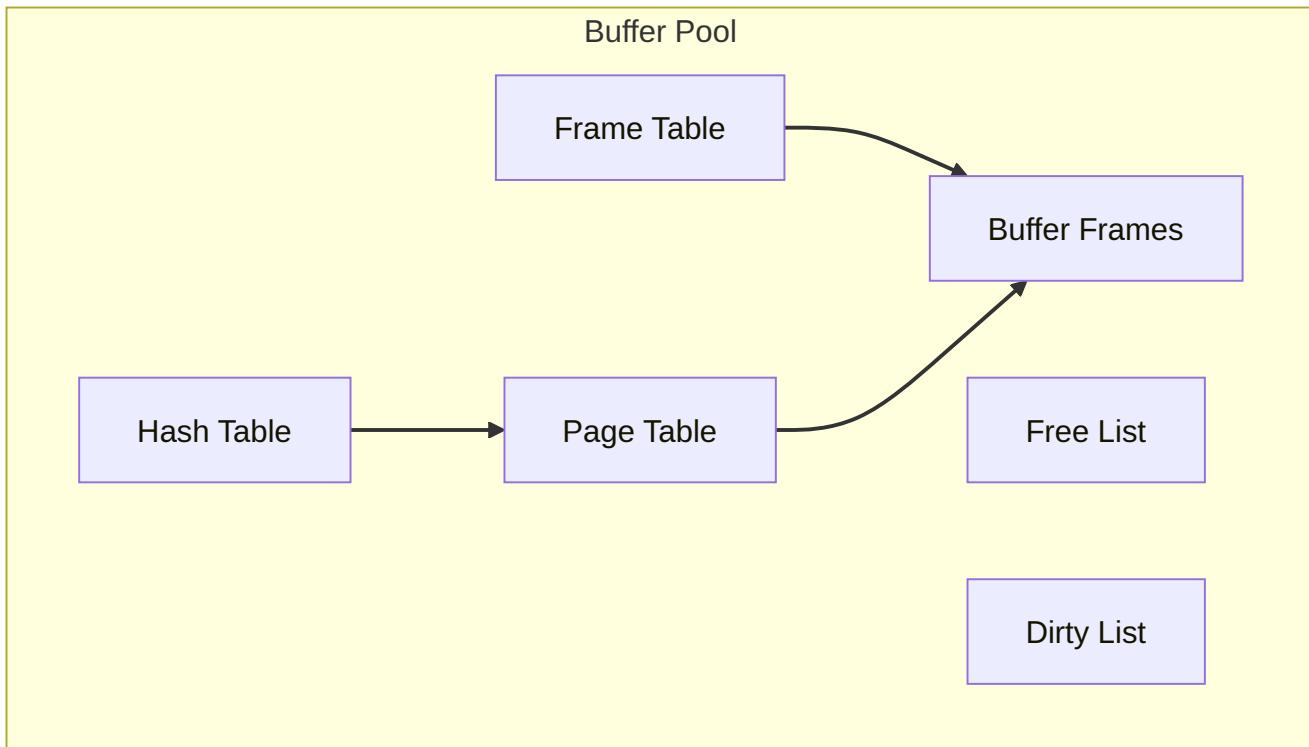
O gerenciamento de buffer é um componente crítico dos sistemas de banco de dados, atuando como intermediário entre a memória principal e o armazenamento em disco.



Arquitetura do Buffer Pool

1. Estruturas Principais

- Frame Table
- Page Table
- Hash Table
- Free List
- Dirty List



2. Componentes de Controle

- Descritores de página
- Contadores de pin
- Bits de estado
- Timestamps

Políticas de Substituição

1. Algoritmos Básicos

- LRU (Least Recently Used)
- Clock
- MRU (Most Recently Used)
- Random

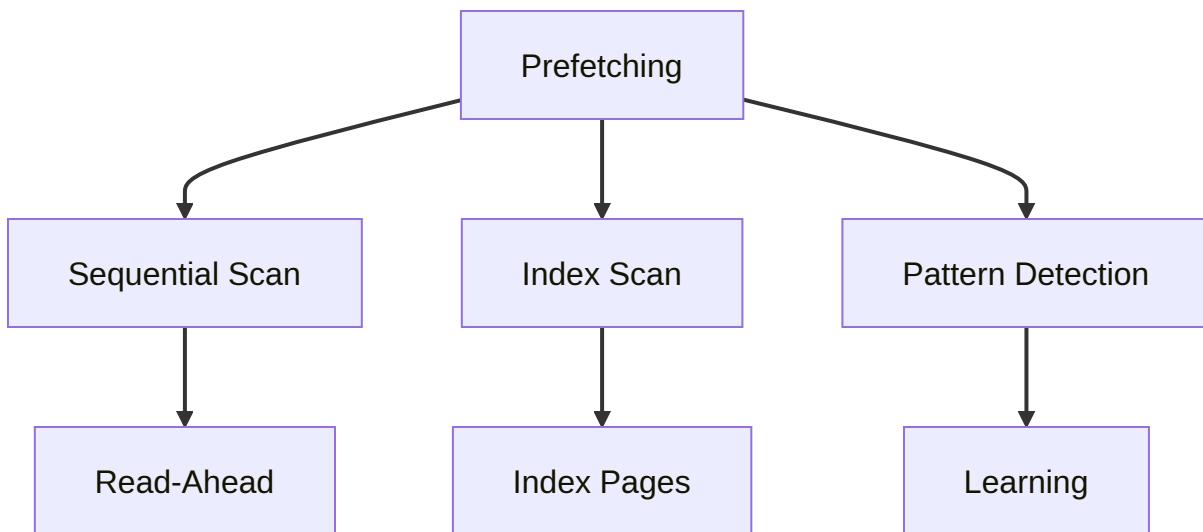
2. Algoritmos Avançados

- LRU-K
- 2Q
- ARC (Adaptive Replacement Cache)
- CLOCK-Pro

Otimizações de Performance

1. Técnicas de Prefetching

- Sequential
- Index-based
- Pattern-based
- Adaptive



2. Write Strategies

- Force/No-Force
- Steal/No-Steal

- Group Commit
- Background Writing

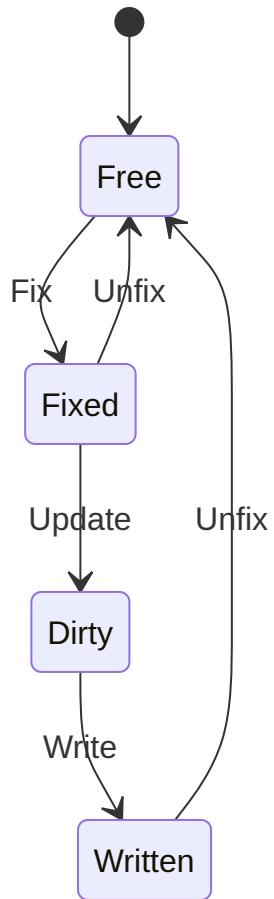
Controle de Concorrência

1. Mecanismos de Latch

- Shared latches
- Exclusive latches
- Latch queuing
- Deadlock prevention

2. Buffer Fix

- Pin count
- Fix duration
- Unfix operations
- Reference counting

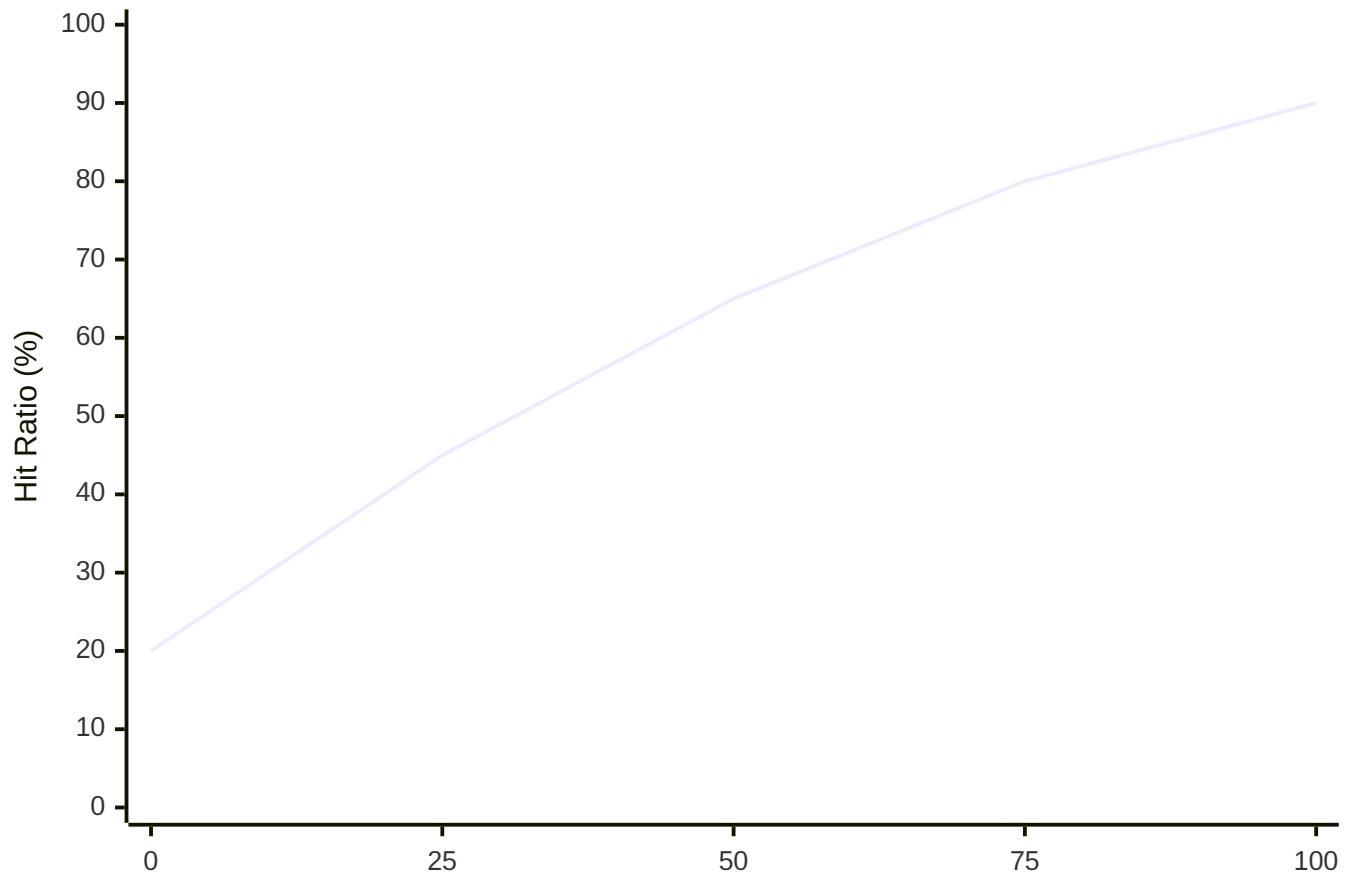


Monitoramento e Diagnóstico

1. Métricas Principais

- Hit ratio
- Buffer utilization
- Write frequency
- Eviction rate

Buffer Pool Performance



2. Ferramentas de Análise

- Buffer pool statistics
- Page access patterns
- I/O monitoring
- Memory pressure

Configuração e Tuning

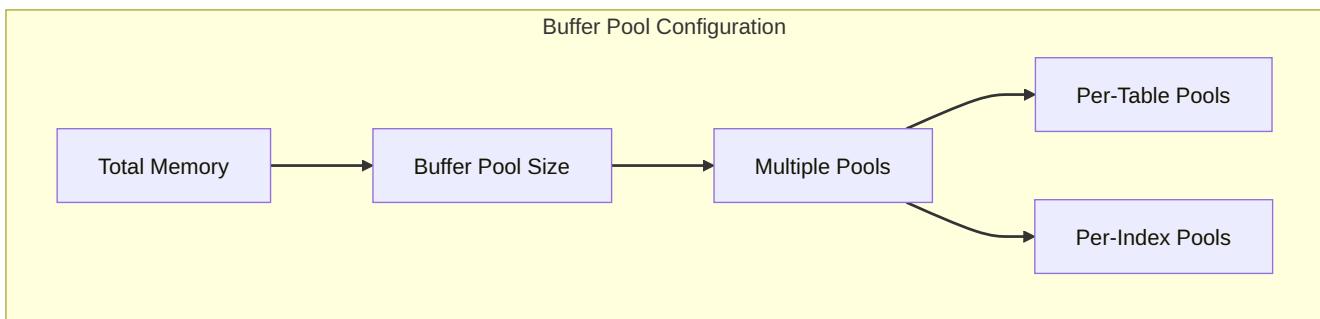
1. Parâmetros Críticos

- Buffer pool size
- Page size

- Number of partitions
- Write threshold

2. Otimizações Específicas

- Multiple buffer pools
- Page compression
- Memory-mapped I/O
- Direct I/O



Recuperação e Consistência

1. Recovery Integration

- Checkpoint processing
- Redo logging
- Undo logging
- Recovery actions

2. Consistency Management

- Page consistency
- Buffer coherency

- Cache invalidation
- Version control

Tendências Modernas

1. Novas Tecnologias

- Non-volatile memory
- Hardware transactional memory
- RDMA-aware buffering
- Smart storage

2. Otimizações Emergentes

- ML-based prediction
- Adaptive algorithms
- Hybrid storage integration
- Cloud-optimized buffering

Conclusão

O gerenciamento eficiente do buffer é fundamental para o desempenho do banco de dados. A escolha e configuração adequada das políticas e mecanismos apresentados impacta diretamente na eficiência do sistema.

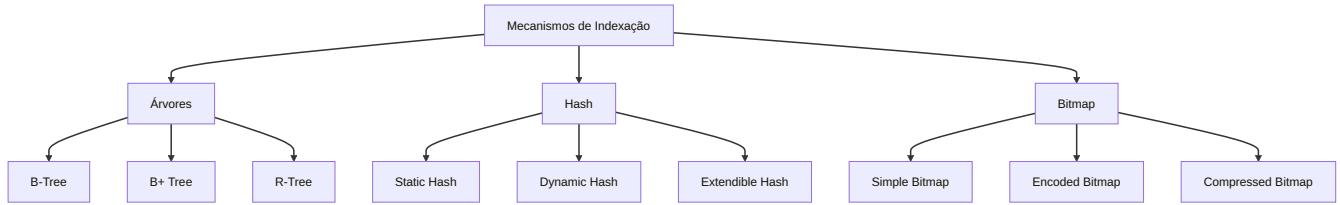
Referências

1. "Database Management Systems" - Ramakrishnan e Gehrke
2. "Transaction Processing: Concepts and Techniques" - Gray e Reuter
3. "PostgreSQL Buffer Management" - Documentation

4. "MySQL InnoDB Buffer Pool" - Technical Documentation

Mecanismos de Indexação

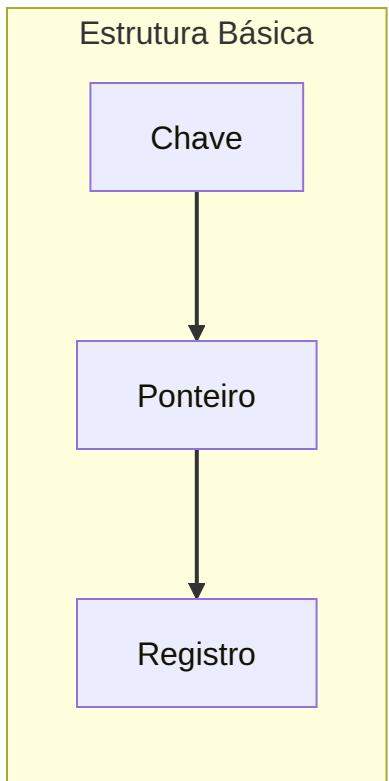
Os mecanismos de indexação são estruturas fundamentais que otimizam o acesso aos dados em sistemas de banco de dados.



Fundamentos de Indexação

1. Conceitos Básicos

- Chaves de busca
- Registros de índice
- Densidade de índice
- Seletividade



2. Classificação

- Primário vs. Secundário
- Denso vs. Esparsos
- Clustered vs. Non-clustered
- Single-level vs. Multi-level

Estruturas de Árvore

1. Árvores B

- Propriedades
- Operações básicas
- Balanceamento
- Split e Merge

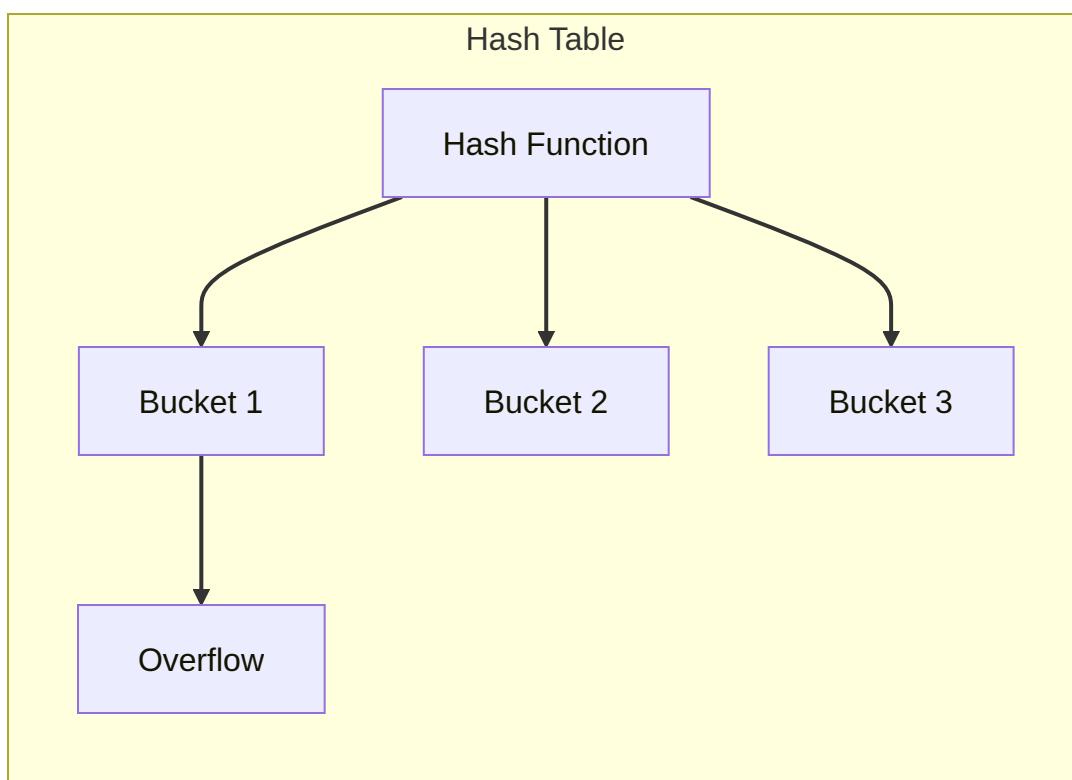
2. Árvores B+

- Estrutura de folhas
- Sequência de folhas
- Range queries
- Bulk loading

Estruturas Hash

1. Hashing Estático

- Funções hash
- Tratamento de colisões
- Fator de carga
- Overflow chains



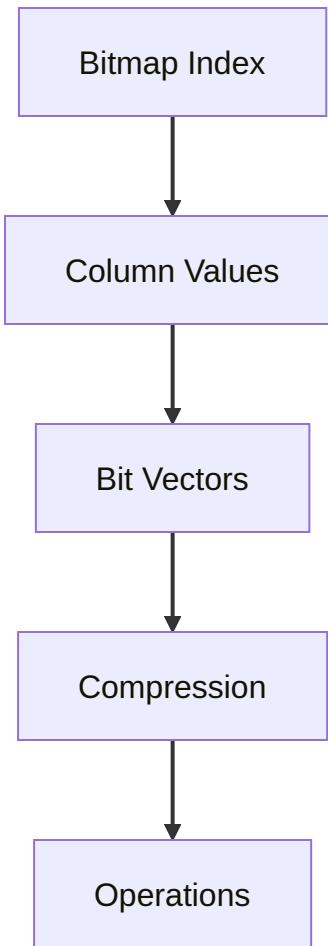
2. Hashing Dinâmico

- Directory structure
- Split operations
- Merge operations
- Directory management

Índices Bitmap

1. Estrutura Básica

- Vetores de bits
- Operações lógicas
- Compressão
- Atualização



2. Otimizações

- Encoding schemes
- Compression techniques
- Cardinality handling
- Update strategies

Técnicas Avançadas

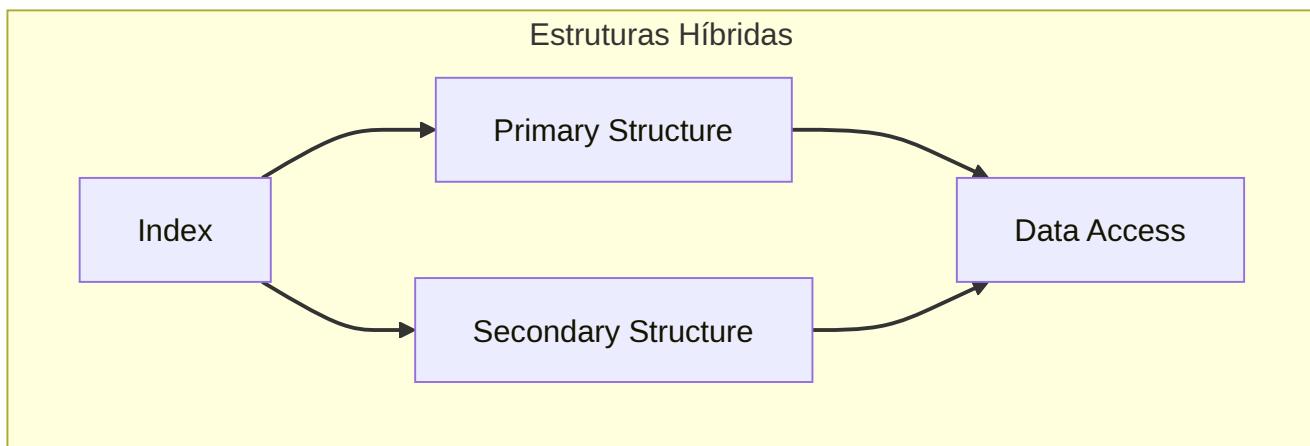
1. Índices Especializados

- Spatial indexes
- Temporal indexes

- Full-text indexes
- JSON indexes

2. Estruturas Híbridas

- Hash-tree combination
- Bitmap-tree indexes
- Multi-dimensional indexes
- Adaptive indexes



Otimização e Manutenção

1. Estratégias de Criação

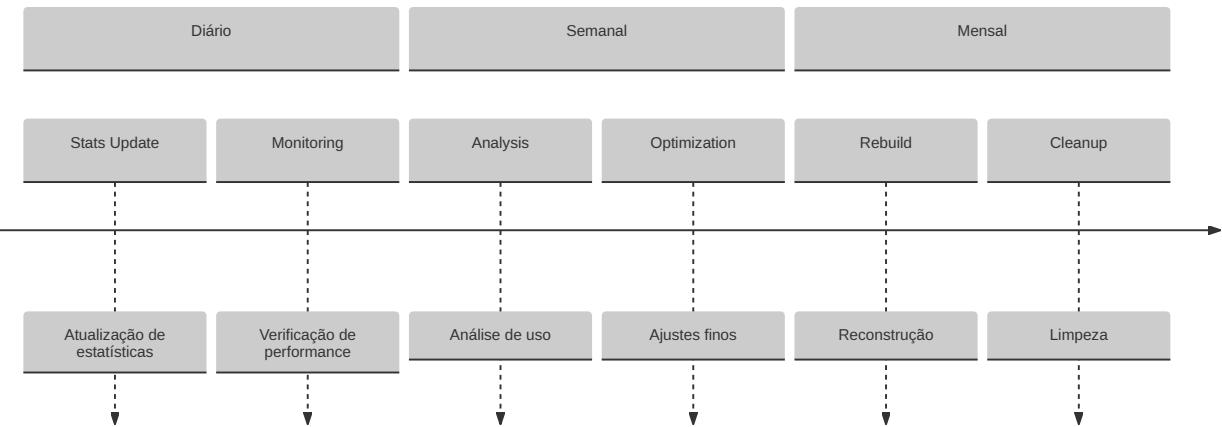
- Index selection
- Key selection
- Storage allocation
- Build optimization

2. Manutenção

- Statistics update

- Reorganization
- Rebuild operations
- Monitoring

Ciclo de Manutenção de Índices

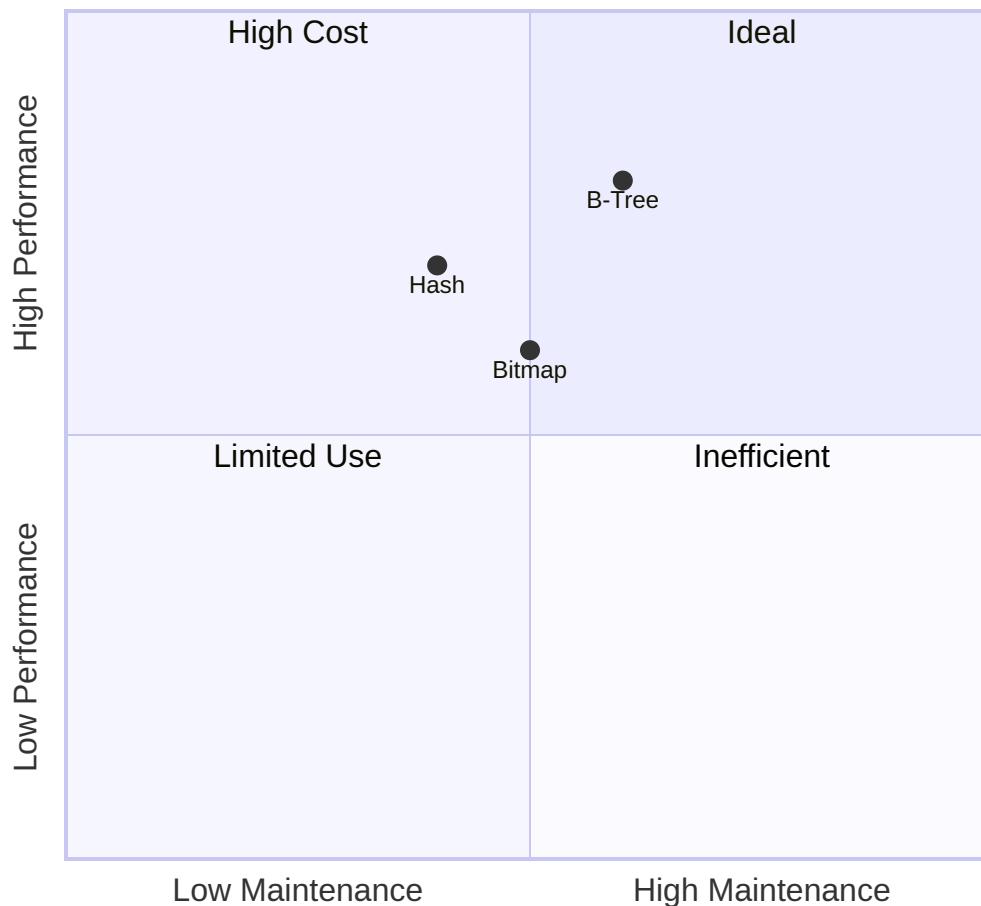


Performance e Trade-offs

1. Métricas de Avaliação

- Access time
- Storage overhead
- Maintenance cost
- Query impact

Trade-offs em Indexação



2. Considerações Práticas

- Workload analysis
- Storage constraints
- Update frequency
- Query patterns

Tendências e Inovações

1. Novas Tecnologias

- Machine learning indexes
- Learned index structures

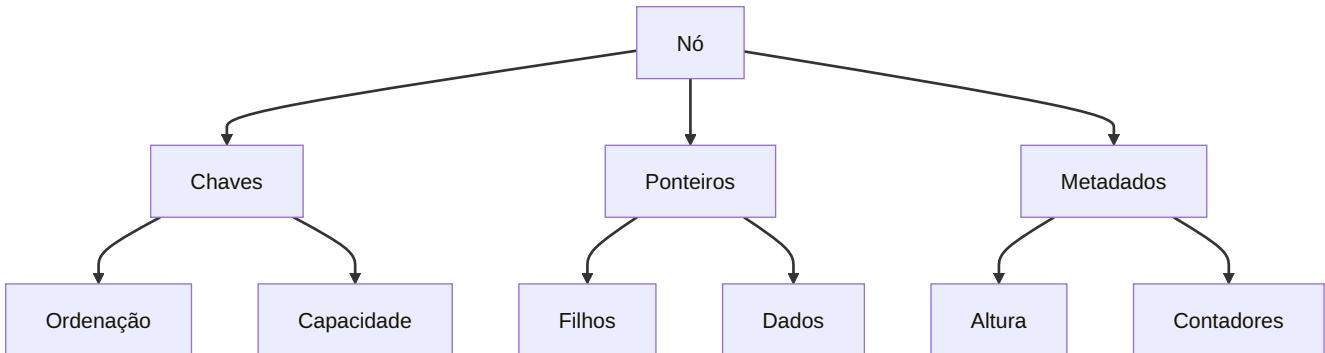
- Hardware-aware indexes
- Cloud-optimized indexes

2. Otimizações Emergentes

- Auto-indexing
- Adaptive indexing
- Predictive maintenance
- Quantum-resistant structures

Implementação de Árvores B

Estrutura Básica



Definição do Nó

```
class BNode {  
    int[] keys;          // array de chaves  
    BNode[] children;   // array de ponteiros  
    int keyCount;       // número de chaves  
    boolean isLeaf;     // flag de folha  
    int minDegree;      // grau mínimo da árvore  
}  
  
class BTree {  
    BNode root;         // raiz da árvore  
    int minDegree;      // grau mínimo da árvore  
  
    public BTree(int degree) {  
        this.root = null;  
        this.minDegree = degree;  
    }  
}
```

Propriedades Fundamentais

- Ordem da árvore (t)

- Número mínimo de chaves ($t-1$)
- Número máximo de chaves ($2t-1$)
- Número mínimo de filhos (t)
- Número máximo de filhos ($2t$)

Operações Fundamentais

1. Busca

Algoritmo de Busca

```
BNode search(BNode node, int key) {
    int i = 0;
    while (i < node.keyCount && key > node.keys[i]) {
        i++;
    }

    if (i < node.keyCount && key == node.keys[i]) {
        return node;
    }

    if (node.isLeaf) {
        return null;
    }

    return search(node.children[i], key);
}
```

Complexidade

- Melhor caso: $O(1)$
- Caso médio: $O(\log n)$
- Pior caso: $O(\log n)$

2. Inserção

Processo de Split

```
void splitChild(BNode parent, int index, BNode child) {  
    BNode newNode = new BNode(child.minDegree);  
    newNode.isLeaf = child.isLeaf;  
    newNode.keyCount = minDegree - 1;  
  
    // Copiar chaves superiores para novo nó  
    for (int j = 0; j < minDegree - 1; j++) {  
        newNode.keys[j] = child.keys[j + minDegree];  
    }  
  
    // Se não for folha, copiar ponteiros correspondentes  
    if (!child.isLeaf) {  
        for (int j = 0; j < minDegree; j++) {  
            newNode.children[j] = child.children[j + minDegree];  
        }  
    }  
  
    child.keyCount = minDegree - 1;  
  
    // Mover ponteiros do pai  
    for (int j = parent.keyCount; j >= index + 1; j--) {  
        parent.children[j + 1] = parent.children[j];  
    }  
  
    parent.children[index + 1] = newNode;  
  
    // Mover chaves do pai e inserir chave mediana  
    for (int j = parent.keyCount - 1; j >= index; j--) {  
        parent.keys[j + 1] = parent.keys[j];  
    }  
    parent.keys[index] = child.keys[minDegree - 1];  
    parent.keyCount++;  
}
```

Algoritmo de Inserção

```
void insert(int key) {  
    if (root == null) {  
        root = new BNode(minDegree);  
        root.keys[0] = key;  
        root.keyCount = 1;  
        root.isLeaf = true;  
    } else {  
        if (root.keyCount == 2 * minDegree - 1) {  
            BNode newRoot = new BNode(minDegree);  
            newRoot.children[0] = root;  
            splitChild(newRoot, 0, root);  
            insertNonFull(newRoot, key);  
            root = newRoot;  
        } else {  
            insertNonFull(root, key);  
        }  
    }  
}
```

3. Remoção

Casos de Remoção

1. Remoção de chave em nó folha
2. Remoção de chave em nó interno
3. Merge de nós
4. Redistribuição de chaves

```
void remove(BNode node, int key) {  
    int idx = findKey(node, key);  
  
    if (idx < node.keyCount && node.keys[idx] == key) {  
        if (node.isLeaf) {
```

```

        removeFromLeaf(node, idx);
    } else {
        removeFromNonLeaf(node, idx);
    }
} else {
    if (node.isLeaf) {
        return; // Chave não encontrada
    }

    boolean flag = (idx == node.keyCount);

    if (node.children[idx].keyCount < minDegree) {
        fill(node, idx);
    }

    if (flag && idx > node.keyCount) {
        remove(node.children[idx - 1], key);
    } else {
        remove(node.children[idx], key);
    }
}
}

```

Otimizações Avançadas

1. Cache-Conscious

Alinhamento de Memória

```

class CacheOptimizedNode {
    private static final int CACHE_LINE_SIZE = 64;
    private long[] keys; // Alinhado em 64 bytes
    private long[] children;

    public CacheOptimizedNode(int degree) {
        keys = new long[2 * degree - 1];
        children = new long[2 * degree];
    }
}

```

```
    }  
}
```

Técnicas de Prefetching

- Software prefetching
- Hardware prefetching hints
- Cache line padding

2. Concorrência

Locks Granulares

```
class ConcurrentBNode {  
    private ReentrantReadWriteLock lock;  
    private volatile boolean isDeleted;  
  
    public void acquireReadLock() {  
        lock.readLock().lock();  
    }  
  
    public void acquireWriteLock() {  
        lock.writeLock().lock();  
    }  
}
```

Versioning

- MVCC (Multi-Version Concurrency Control)
- Version chains
- Garbage collection

Variantes de Implementação

1. Copy-on-Write

```

class COWBNode {
    private final int[] keys;
    private final BNode[] children;
    private final AtomicReference<COWBNode> next;

    public COWBNode copy() {
        COWBNode newNode = new COWBNode(keys.clone(),
                                         children.clone());
        return newNode;
    }
}

```

2. Bulk Loading

Algoritmo Bottom-up

```

void bulkLoad(int[] sortedKeys) {
    int leafSize = 2 * minDegree - 1;
    List<BNode> leaves = new ArrayList<>();

    // Criar nós folha
    for (int i = 0; i < sortedKeys.length; i += leafSize) {
        BNode leaf = new BNode(minDegree);
        leaf.isLeaf = true;
        int count = Math.min(leafSize,
                            sortedKeys.length - i);
        System.arraycopy(sortedKeys, i,
                        leaf.keys, 0, count);
        leaf.keyCount = count;
        leaves.add(leaf);
    }

    // Construir níveis superiores
    buildUpperLevels(leaves);
}

```

Estruturas de Suporte

1. Buffer Management

Política de Cache

```
class BufferPool {  
    private final int capacity;  
    private final Map<Long, BNode> pages;  
    private final LRUcache<Long> lru;  
  
    public BNode getPage(long pageId) {  
        BNode page = pages.get(pageId);  
        if (page != null) {  
            lru.access(pageId);  
            return page;  
        }  
        return loadFromDisk(pageId);  
    }  
}
```

2. Recovery

Write-Ahead Logging

```
class LogRecord {  
    enum Type { INSERT, DELETE, SPLIT, MERGE }  
    private final Type type;  
    private final long pageId;  
    private final int key;  
    private final byte[] beforeImage;  
    private final byte[] afterImage;  
}
```

Aspectos Práticos

1. Monitoramento

Métricas Chave

- Altura da árvore
- Fator de ocupação
- Taxa de split/merge
- Latência de operações

2. Manutenção

Rebalanceamento Adaptativo

```
void rebalance(BNode node) {  
    if (node.keyCount < minDegree - 1) {  
        mergeOrRedistribute(node);  
    } else if (node.keyCount > 2 * minDegree - 1) {  
        split(node);  
    }  
  
    if (!node.isLeaf) {  
        for (int i = 0; i <= node.keyCount; i++) {  
            rebalance(node.children[i]);  
        }  
    }  
}
```

Estruturas de Dados Auxiliares

1. Iterator

```
class BTreeIterator implements Iterator<Integer> {  
    private final Stack<BNode> path;  
    private final Stack<Integer> indices;  
  
    public boolean hasNext() {  
        return !path.isEmpty();  
    }
```

```

public Integer next() {
    BNode current = path.peek();
    int idx = indices.peek();

    int key = current.keys[idx];
    advanceToNext();
    return key;
}

}

```

2. Range Scan

```

List<Integer> rangeSearch(int start, int end) {
    List<Integer> result = new ArrayList<>();
    rangeSearchRecursive(root, start, end, result);
    return result;
}

void rangeSearchRecursive(BNode node, int start,
                         int end, List<Integer> result) {
    int i = 0;

    while (i < node.keyCount && node.keys[i] < start) {
        i++;
    }

    while (i < node.keyCount && node.keys[i] <= end) {
        if (!node.isLeaf) {
            rangeSearchRecursive(node.children[i],
                                 start, end, result);
        }
        result.add(node.keys[i]);
        i++;
    }

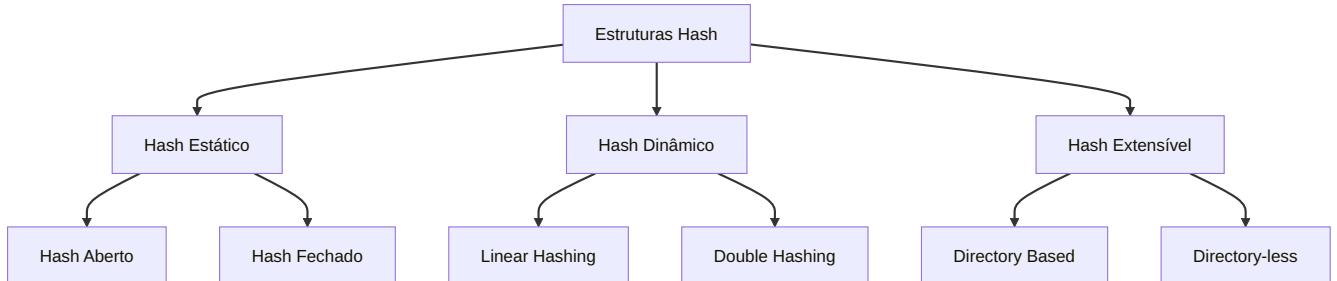
    if (!node.isLeaf && i <= node.keyCount) {

```

```
        rangeSearchRecursive(node.children[i],  
                             start, end, result);  
    }  
}
```

Estruturas Hash

Fundamentos



1. Funções Hash

```
class HashFunction {  
    // Multiplicação  
    long multiplyHash(String key, int tableSize) {  
        long hash = 0;  
        for (char c : key.toCharArray()) {  
            hash = 31 * hash + c;  
        }  
        return Math.abs(hash % tableSize);  
    }  
  
    // FNV Hash  
    long fnvHash(byte[] data) {  
        long hash = 0xcbf29ce484222325L;  
        for (byte b : data) {  
            hash *= 0x100000001b3L;  
            hash ^= b;  
        }  
        return hash;  
    }  
}
```

2. Tratamento de Colisões

Encadeamento Externo

```

class Node<K,V> {
    K key;
    V value;
    Node<K,V> next;

    Node(K key, V value) {
        this.key = key;
        this.value = value;
    }
}

class HashTable<K,V> {
    private Node<K,V>[] table;
    private int size;

    @SuppressWarnings("unchecked")
    public HashTable(int capacity) {
        table = (Node<K,V>[]) new Node[capacity];
        size = 0;
    }

    public void put(K key, V value) {
        int index = hash(key);
        Node<K,V> node = table[index];

        while (node != null) {
            if (node.key.equals(key)) {
                node.value = value;
                return;
            }
            node = node.next;
        }

        Node<K,V> newNode = new Node<>(key, value);
        newNode.next = table[index];
        table[index] = newNode;
        size++;
    }
}

```

```
    }  
}
```

Endereçamento Aberto

```
class OpenAddressingHash<K, V> {  
    private Entry<K, V>[] table;  
    private int size;  
    private static final double LOAD_FACTOR = 0.75;  
  
    private static class Entry<K, V> {  
        K key;  
        V value;  
        boolean isDeleted;  
  
        Entry(K key, V value) {  
            this.key = key;  
            this.value = value;  
        }  
    }  
  
    public V get(K key) {  
        int index = findKey(key);  
        return index != -1 ? table[index].value : null;  
    }  
  
    private int findKey(K key) {  
        int hash = hash(key);  
        int i = 0;  
  
        while (i < table.length) {  
            int j = (hash + probe(i)) % table.length;  
  
            if (table[j] == null) return -1;  
            if (!table[j].isDeleted &&  
                table[j].key.equals(key)) {  
                return j;  
            }  
        }  
    }  
}
```

```

        }
        i++;
    }
    return -1;
}

private int probe(int i) {
    return i * i; // Quadratic probing
}
}

```

Hash Dinâmico

1. Linear Hashing

```

class LinearHash<K,V> {
    private ArrayList<Bucket<K,V>> buckets;
    private int splitPointer;
    private int level;
    private double loadFactor;

    private static class Bucket<K,V> {
        Map<K,V> entries;
        int localDepth;

        Bucket(int depth) {
            entries = new HashMap<>();
            localDepth = depth;
        }
    }

    public void insert(K key, V value) {
        int bucketIndex = getBucketIndex(key);
        Bucket<K,V> bucket = buckets.get(bucketIndex);

        bucket.entries.put(key, value);
    }
}

```

```

        if (shouldSplit()) {
            split();
        }

    private void split() {
        Bucket<K,V> oldBucket = buckets.get(splitPointer);
        Bucket<K,V> newBucket = new Bucket<>(level);

        // Redistribuir entradas
        Map<K,V> oldEntries = oldBucket.entries;
        oldBucket.entries = new HashMap<>();

        for (Map.Entry<K,V> entry : oldEntries.entrySet()) {
            int newIndex = getBucketIndex(entry.getKey());
            if (newIndex == splitPointer) {
                oldBucket.entries.put(entry.getKey(),
                                      entry.getValue());
            } else {
                newBucket.entries.put(entry.getKey(),
                                      entry.getValue());
            }
        }

        buckets.add(newBucket);
        splitPointer++;
    }

    if (splitPointer == Math.pow(2, level)) {
        splitPointer = 0;
        level++;
    }
}
}

```

2. Extendible Hashing

```

class ExtendibleHash<K,V> {
    private Directory<K,V> directory;
    private int globalDepth;

    private static class Directory<K,V> {
        Bucket<K,V>[] buckets;
        int size;

        @SuppressWarnings("unchecked")
        Directory(int size) {
            this.size = size;
            buckets = new Bucket[size];
        }
    }

    private static class Bucket<K,V> {
        Map<K,V> entries;
        int localDepth;
        static final int CAPACITY = 4;

        Bucket(int depth) {
            entries = new HashMap<>();
            localDepth = depth;
        }

        boolean isFull() {
            return entries.size() >= CAPACITY;
        }
    }

    public void insert(K key, V value) {
        int dirIndex = hash(key) & ((1 << globalDepth) - 1);
        Bucket<K,V> bucket = directory.buckets[dirIndex];

        if (bucket.isFull()) {
            if (bucket.localDepth == globalDepth) {
                doubleDirectory();
            }
        }
    }
}

```

```

        }
        split(dirIndex);
        insert(key, value);
    } else {
        bucket.entries.put(key, value);
    }
}

private void split(int bucketIndex) {
    Bucket<K,V> oldBucket = directory.buckets[bucketIndex];
    Bucket<K,V> newBucket = new Bucket<>(oldBucket.localDepth
+ 1);

    Map<K,V> oldEntries = oldBucket.entries;
    oldBucket.entries = new HashMap<>();
    oldBucket.localDepth++;

    int mask = 1 << (oldBucket.localDepth - 1);
    for (Map.Entry<K,V> entry : oldEntries.entrySet()) {
        int newIndex = hash(entry.getKey()) & ((1 <<
oldBucket.localDepth) - 1);
        if ((newIndex & mask) == 0) {
            oldBucket.entries.put(entry.getKey(),
entry.getValue());
        } else {
            newBucket.entries.put(entry.getKey(),
entry.getValue());
        }
    }

    // Atualizar diretório
    for (int i = 0; i < directory.size; i++) {
        if (directory.buckets[i] == oldBucket && (i & mask) !=
0) {
            directory.buckets[i] = newBucket;
        }
    }
}

```

```
    }  
}
```

Otimizações

1. Cache-Conscious Hashing

```
class CacheOptimizedHash<K, V> {  
    private static final int CACHE_LINE_SIZE = 64;  
    private static final int ENTRIES_PER_BUCKET =  
        CACHE_LINE_SIZE / (8 + 8); // key + value ptr  
  
    private static class Bucket<K, V> {  
        long[] keys;  
        V[] values;  
        int size;  
  
        @SuppressWarnings("unchecked")  
        Bucket() {  
            keys = new long[ENTRIES_PER_BUCKET];  
            values = (V[]) new Object[ENTRIES_PER_BUCKET];  
        }  
    }  
}
```

2. Concurrent Hashing

```
class ConcurrentHash<K, V> {  
    private static final int SHARD_COUNT = 16;  
    private final HashTable<K, V>[] shards;  
    private final ReentrantLock[] locks;  
  
    @SuppressWarnings("unchecked")  
    public ConcurrentHash() {  
        shards = new HashTable[SHARD_COUNT];  
        locks = new ReentrantLock[SHARD_COUNT];  
    }
```

```

        for (int i = 0; i < SHARD_COUNT; i++) {
            shards[i] = new HashTable<>();
            locks[i] = new ReentrantLock();
        }
    }

    public V put(K key, V value) {
        int shardIndex = getShard(key);
        locks[shardIndex].lock();
        try {
            return shards[shardIndex].put(key, value);
        } finally {
            locks[shardIndex].unlock();
        }
    }

    private int getShard(K key) {
        return Math.abs(key.hashCode() % SHARD_COUNT);
    }
}

```

Estruturas Especializadas

1. Bloom Filter

```

class BloomFilter<T> {
    private BitSet bitset;
    private int size;
    private int numHashFunctions;
    private HashFunction[] hashFunctions;

    public BloomFilter(int size, int numHash) {
        this.size = size;
        this.numHashFunctions = numHash;
        this.bitset = new BitSet(size);
        this.hashFunctions = new HashFunction[numHash];
    }
}

```

```

        for (int i = 0; i < numHash; i++) {
            hashFunctions[i] = new HashFunction(i);
        }
    }

    public void add(T item) {
        for (HashFunction hf : hashFunctions) {
            bitset.set(hf.hash(item) % size);
        }
    }

    public boolean mightContain(T item) {
        for (HashFunction hf : hashFunctions) {
            if (!bitset.get(hf.hash(item) % size)) {
                return false;
            }
        }
        return true;
    }
}

```

2. Cuckoo Hashing

```

class CuckooHash<K,V> {
    private static final int MAX_LOOP = 100;
    private Entry<K,V>[][] tables;
    private HashFunction[] hashFunctions;

    private static class Entry<K,V> {
        K key;
        V value;

        Entry(K key, V value) {
            this.key = key;
            this.value = value;
        }
    }
}

```

```

}

@SuppressWarnings("unchecked")
public CuckooHash(int capacity) {
    tables = new Entry[2][capacity];
    hashFunctions = new HashFunction[]{
        new HashFunction(0),
        new HashFunction(1)
    };
}

public boolean insert(K key, V value) {
    Entry<K,V> entry = new Entry<>(key, value);

    for (int i = 0; i < MAX_LOOP; i++) {
        for (int j = 0; j < 2; j++) {
            int pos = hashFunctions[j].hash(key) %
tables[j].length;
            Entry<K,V> temp = tables[j][pos];
            tables[j][pos] = entry;

            if (temp == null) return true;
            entry = temp;
        }
    }

    // Rehash needed
    return false;
}
}

```

Monitoramento e Manutenção

1. Métricas

```

class HashMetrics {
    private long collisions;
}

```

```

private long resizes;
private double loadFactor;
private long[] bucketSizes;

public void recordCollision() {
    collisions++;
}

public void recordResize() {
    resizes++;
}

public void updateLoadFactor(int entries, int capacity) {
    loadFactor = (double) entries / capacity;
}

public String getStats() {
    return String.format(
        "Collisions: %d\nResizes: %d\nLoad Factor: %.2f",
        collisions, resizes, loadFactor
    );
}

}

```

2. Auto-tuning

```

class AdaptiveHash<K,V> {
    private static final double RESIZE_THRESHOLD = 0.75;
    private static final double COLLISION_THRESHOLD = 0.1;

    private HashTable<K,V> table;
    private HashMetrics metrics;

    public void tune() {
        if (metrics.getLoadFactor() > RESIZE_THRESHOLD) {
            resize(table.capacity() * 2);
        }
    }
}

```

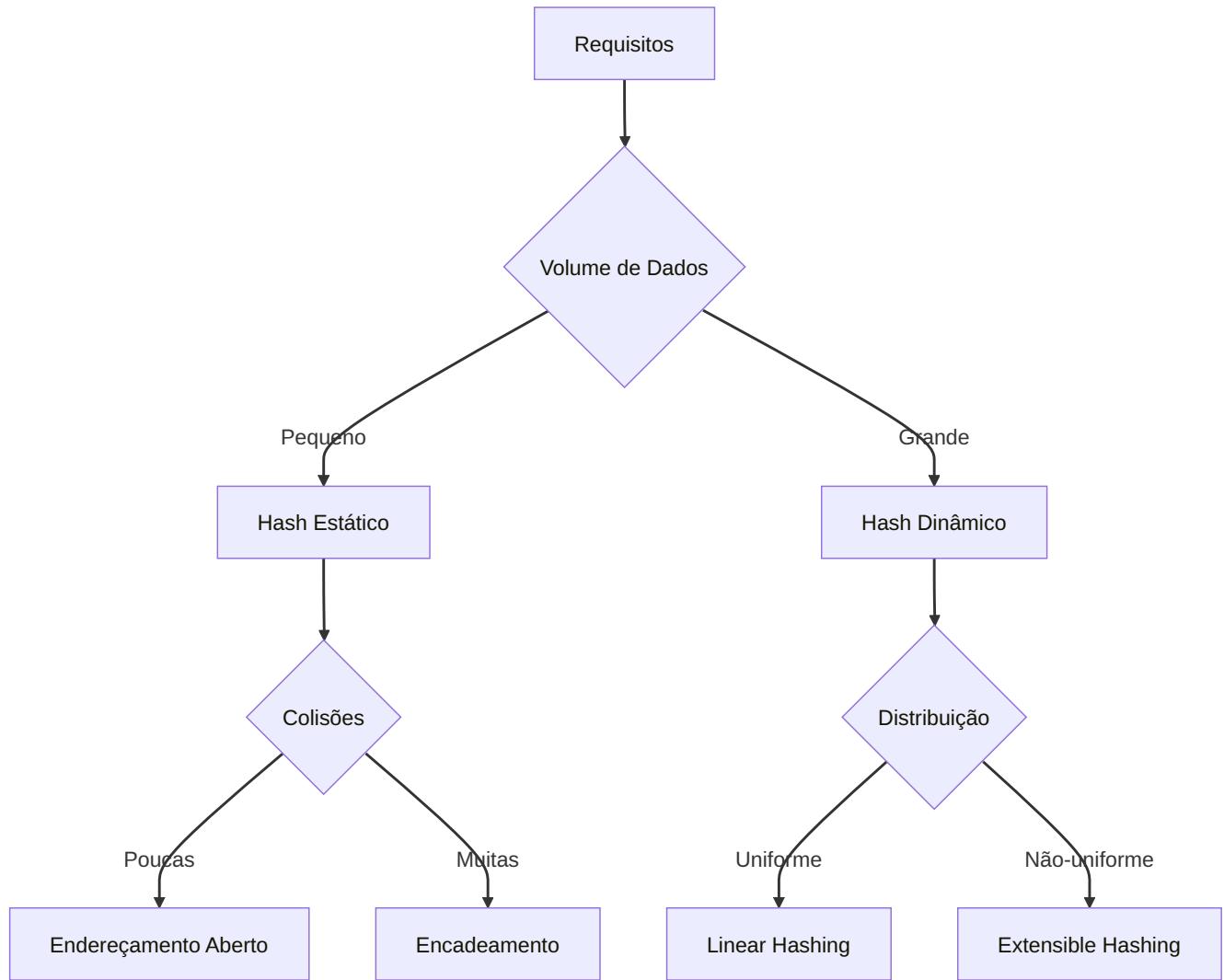
```
    if (metrics.getCollisionRate() > COLLISION_THRESHOLD) {
        changeHashFunction();
    }
}

private void changeHashFunction() {
    HashFunction[] candidates = {
        new MultiplyHash(),
        new FNVHash(),
        new MurmurHash()
    };

    // Avaliar e selecionar a melhor função
    HashFunction best = evaluateHashFunctions(candidates);
    table.setHashFunction(best);
}
}
```

Considerações Práticas

1. Escolha da Estrutura

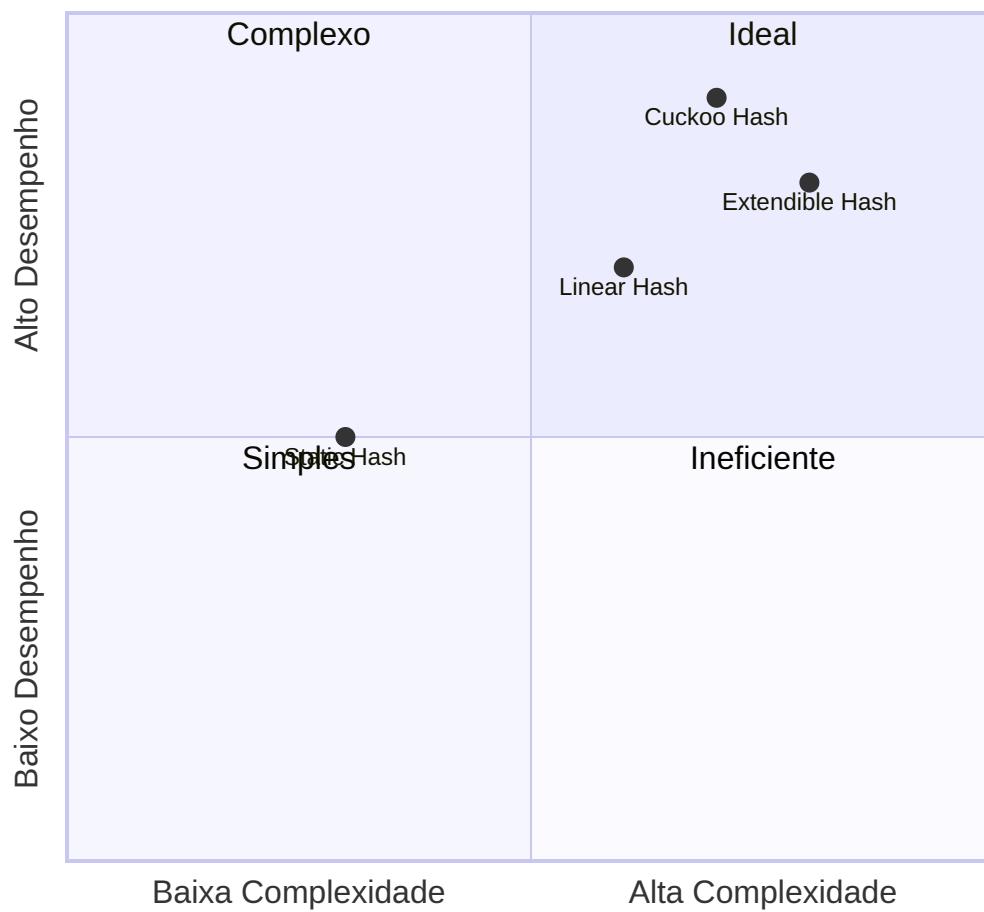


2. Trade-offs

- Memória vs. Velocidade
- Complexidade vs. Flexibilidade

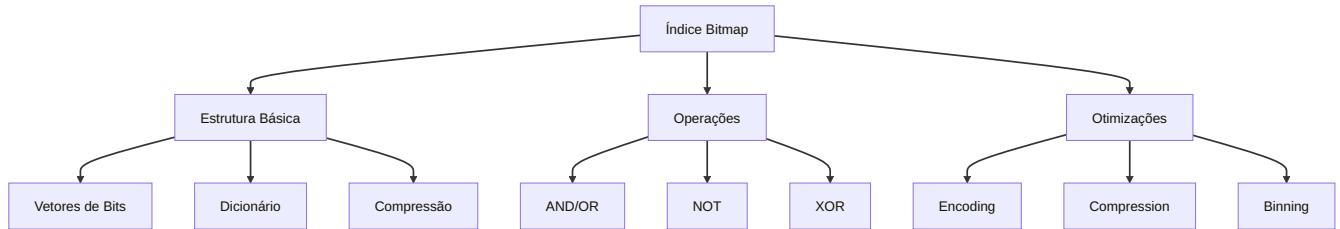
- Concorrência vs. Consistência
- Localidade vs. Distribuição

Trade-offs em Estruturas Hash



Índices Bitmap

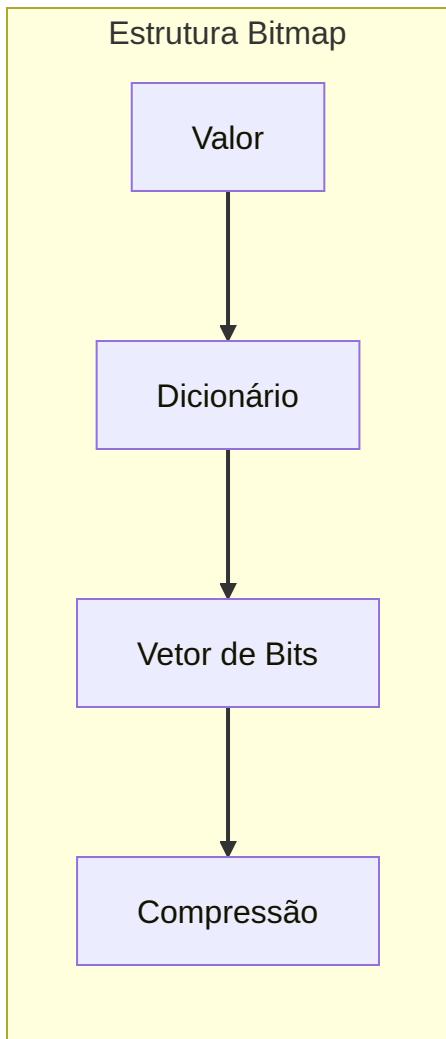
Os índices bitmap são estruturas especializadas que utilizam vetores de bits para representar a presença ou ausência de valores em colunas, sendo particularmente eficientes para colunas com baixa cardinalidade.



Fundamentos

1. Estrutura Básica

- Mapeamento valor-bit
- Vetores binários
- Dicionário de valores
- Metadata



2. Tipos de Bitmap

- Simple bitmap
- Encoded bitmap
- Compressed bitmap
- Hierarchical bitmap

Operações Fundamentais

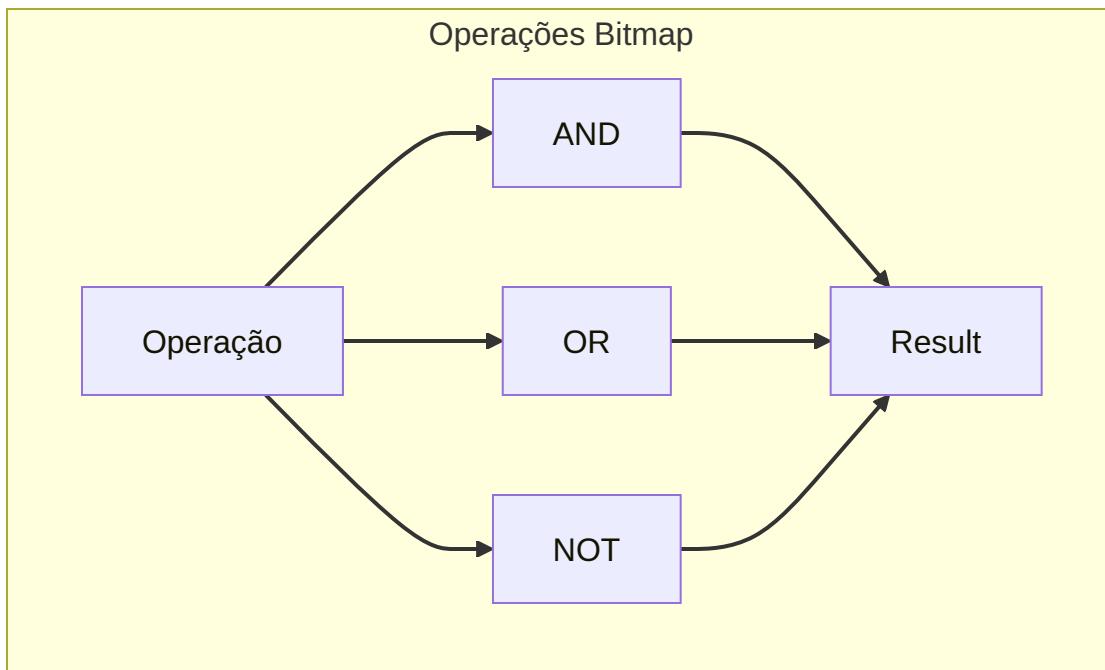
1. Operações Lógicas

- AND (Interseção)

- OR (União)
- NOT (Complemento)
- XOR (Diferença simétrica)

2. Manipulação

- Set bit
- Clear bit
- Flip bit
- Count bits



Otimizações

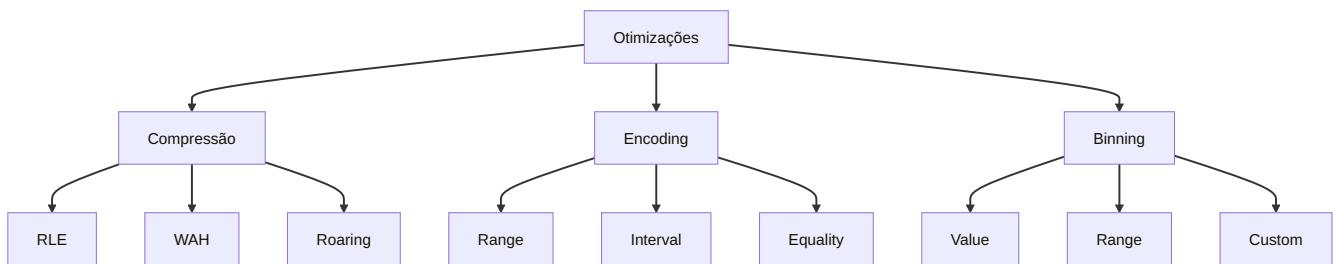
1. Técnicas de Compressão

- Run-length encoding
- Word-aligned hybrid

- Roaring bitmap
- EWAH compression

2. Estratégias de Encoding

- Range encoding
- Interval encoding
- Equality encoding
- Range-equality encoding



Casos de Uso

1. Cenários Ideais

- Baixa cardinalidade
- Consultas analíticas
- Operações em lote
- Data warehousing

2. Limitações

- Alta cardinalidade
- Frequentes atualizações
- Restrições de memória

- Overhead de manutenção

Performance e Otimização

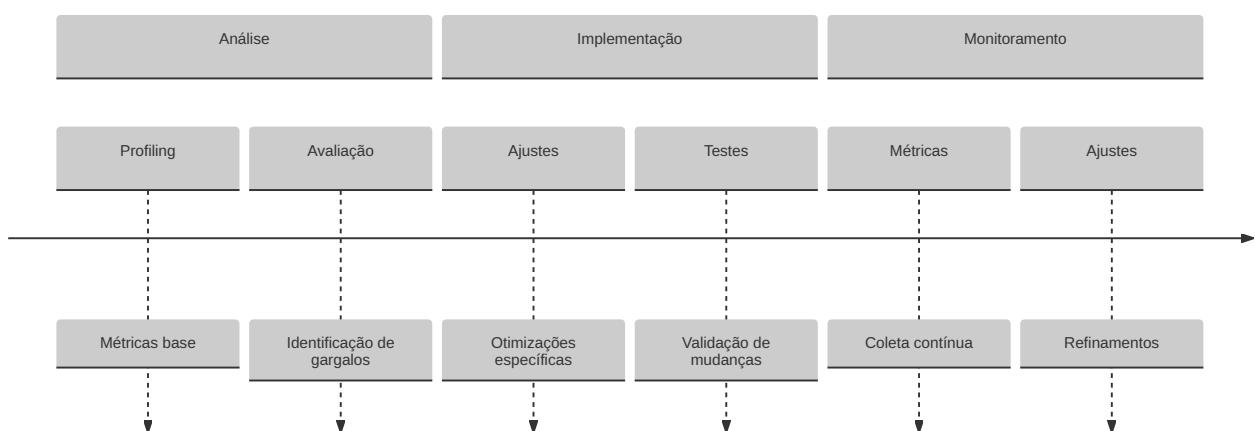
1. Métricas de Avaliação

- Densidade do bitmap
- Taxa de compressão
- Tempo de resposta
- Overhead de memória

2. Estratégias de Otimização

- Binning
- Particionamento
- Caching
- Paralelização

Ciclo de Otimização



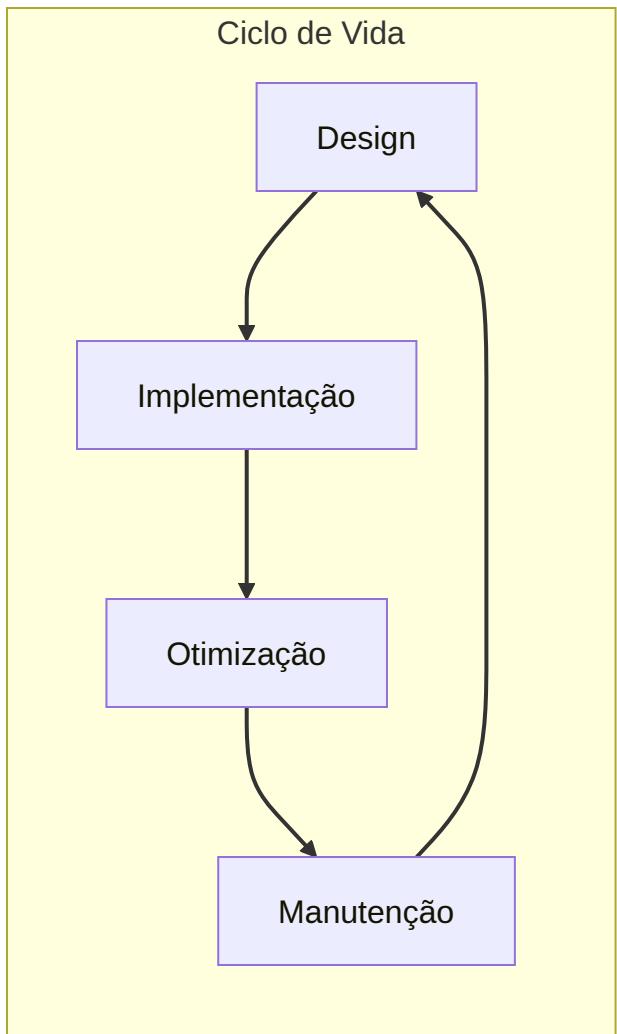
Implementação Prática

1. Considerações de Design

- Estrutura de armazenamento
- Estratégias de atualização
- Gerenciamento de memória
- Concorrência

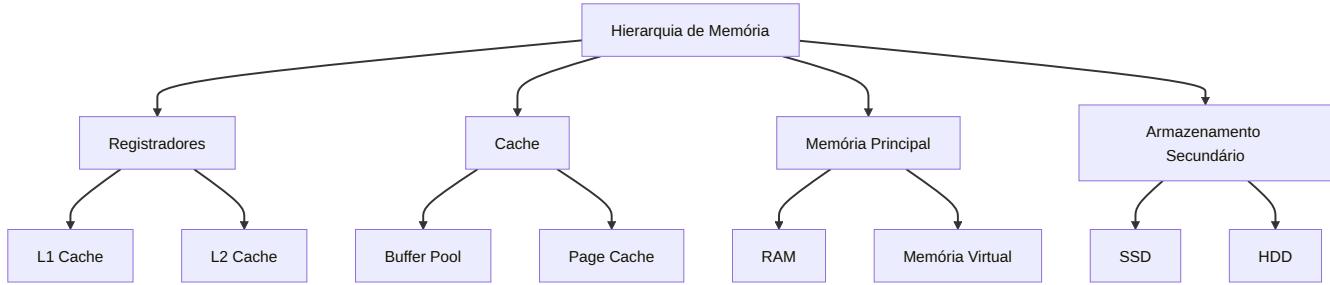
2. Manutenção

- Reconstrução
- Compactação
- Estatísticas
- Monitoramento



Hierarquia de Memória

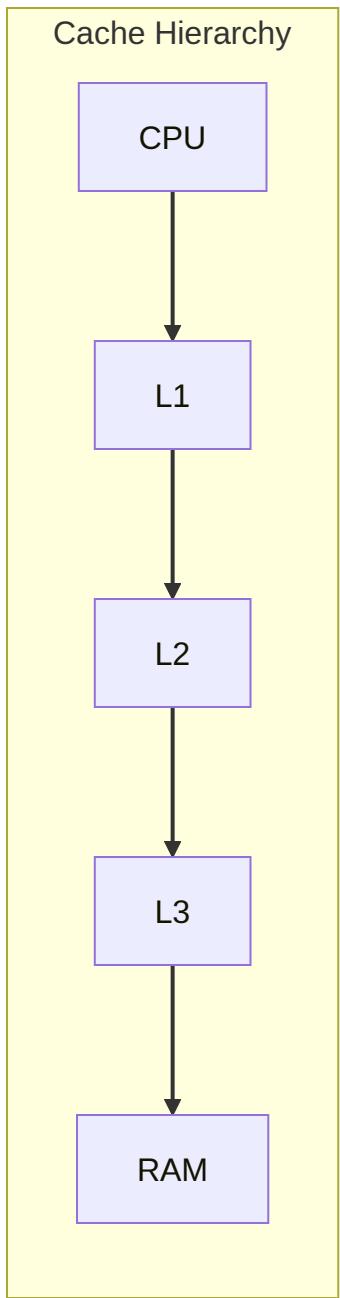
A hierarquia de memória é um conceito fundamental em sistemas de banco de dados que organiza diferentes níveis de armazenamento baseados em velocidade, custo e capacidade.



Níveis de Hierarquia

1. Memória Primária

- **Registradores**
 - Velocidade: < 1ns
 - Capacidade: KB
 - Volatilidade: Sim
 - Custo: Muito Alto
- **Cache**
 - L1/L2/L3
 - Latência: 1-10ns
 - Capacidade: MB
 - Política de substituição



2. Memória Principal

- RAM
 - Acesso direto
 - Latência: ~100ns
 - Capacidade: GB

- Gerenciamento dinâmico
- **Memória Virtual**
 - Paginação
 - Swapping
 - Page tables
 - TLB (Translation Lookaside Buffer)

3. Armazenamento Secundário

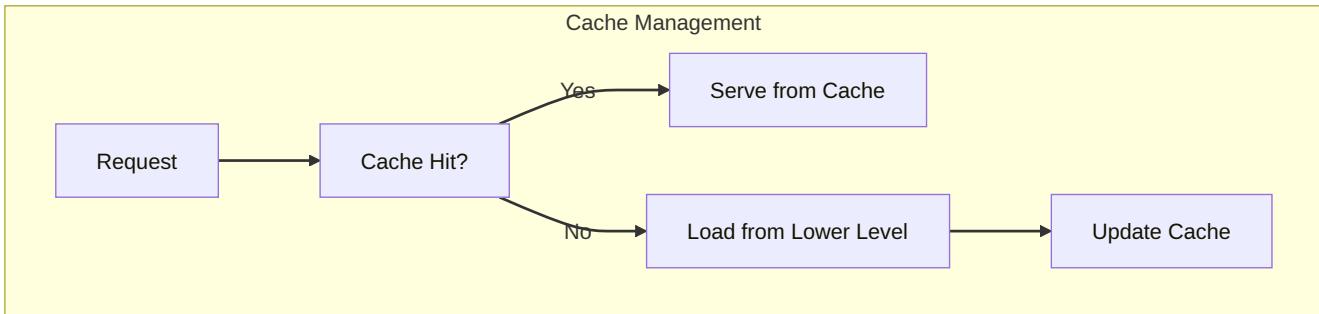
- SSD
 - Flash storage
 - Latência: μ s
 - Wear leveling
 - TRIM support
- HDD
 - Discos magnéticos
 - Latência: ms
 - Fragmentação
 - Seek time

Estratégias de Gerenciamento

1. Políticas de Cache

- LRU (Least Recently Used)
- MRU (Most Recently Used)
- CLOCK

- ARC (Adaptive Replacement Cache)



2. Buffer Management

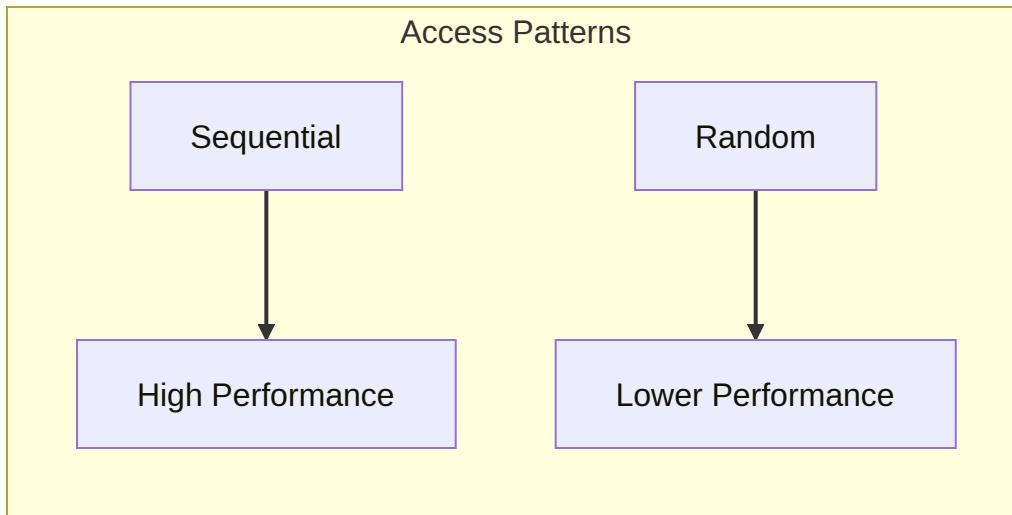
- Políticas de Substituição
 - Page replacement
 - Dirty page handling
 - Prefetching
 - Write-back/Write-through
- Otimizações
 - Sequential prefetch
 - Random prefetch
 - Buffer pool partitioning
 - Multiple buffer pools

Otimização de Performance

1. Técnicas de Otimização

- Locality of Reference
 - Temporal locality
 - Spatial locality

- Sequential access
- Random access



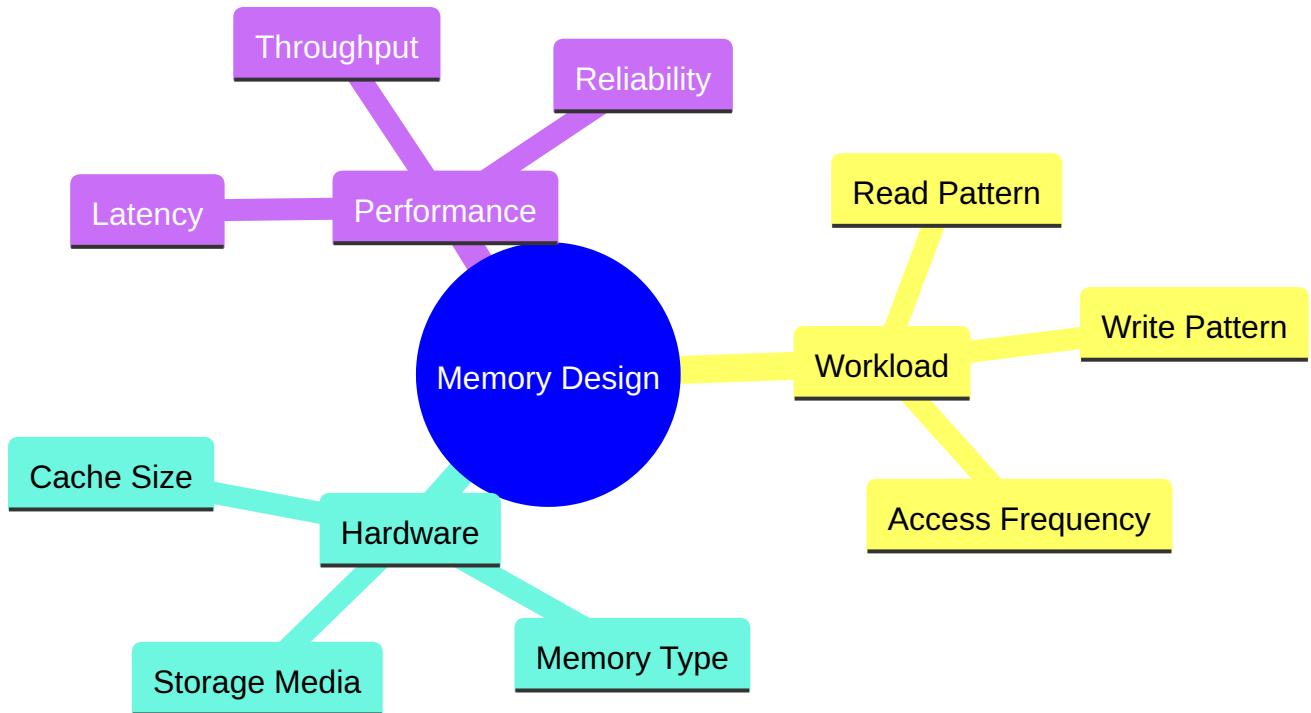
2. Monitoramento e Tuning

- Métricas Chave
 - Hit ratio
 - Miss ratio
 - Response time
 - Throughput
- Ferramentas
 - Performance counters
 - Memory profilers
 - I/O statistics
 - Cache analytics

Considerações Práticas

1. Design Considerations

- Workload Analysis
 - Read/write patterns
 - Access frequency
 - Data volume
 - Concurrency requirements



2. Implementation Guidelines

- Best Practices
 - Memory alignment
 - Cache-conscious data structures
 - Memory barriers
 - NUMA awareness
- Common Pitfalls
 - Cache thrashing

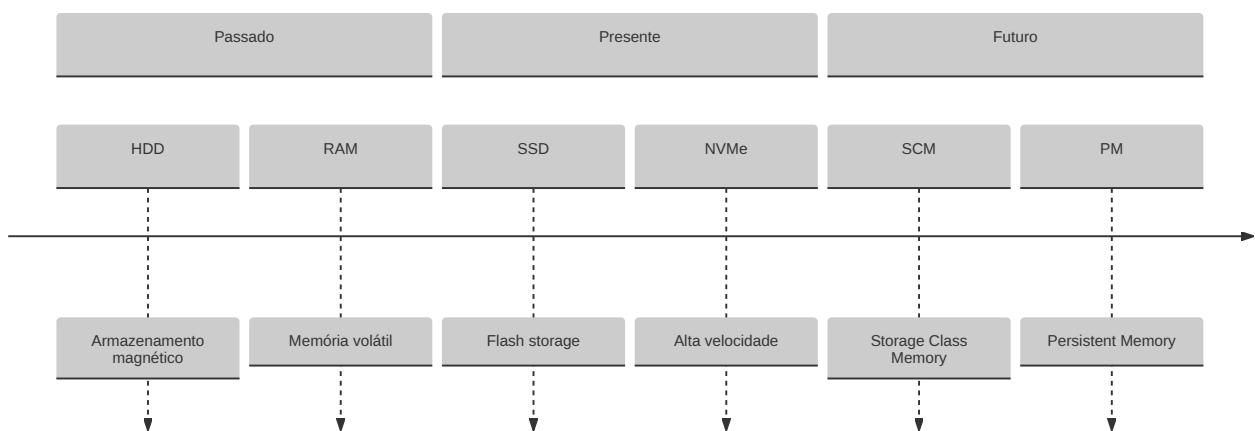
- Memory leaks
- False sharing
- Fragmentation

Tendências e Inovações

1. Emerging Technologies

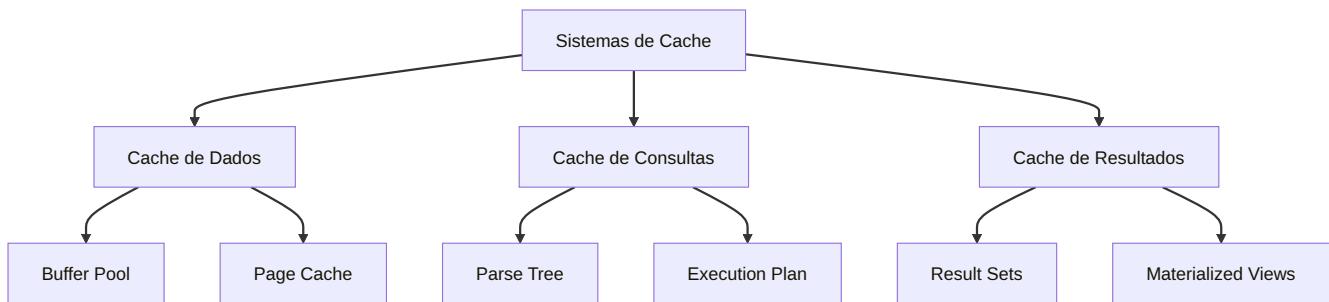
- Persistent Memory
 - NVDIMM
 - Intel Optane
 - Storage Class Memory
- New Architectures
 - In-memory databases
 - Hybrid memory systems
 - Disaggregated memory

Evolução da Hierarquia de Memória



Sistemas de Cache

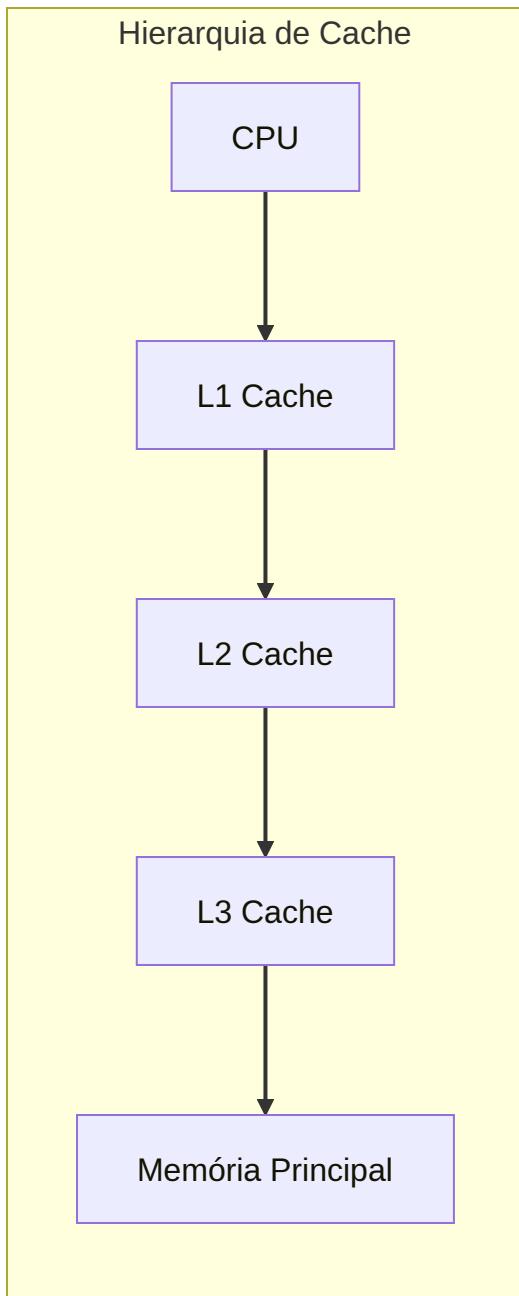
Os sistemas de cache são componentes críticos que otimizam o acesso a dados, reduzindo a latência e melhorando o desempenho geral do sistema de banco de dados.



Arquitetura de Cache

1. Níveis de Cache

- Cache L1/L2/L3
 - Hierarquia
 - Latência
 - Capacidade
 - Políticas



2. Buffer Pool

- Estrutura interna
- Gestão de páginas
- Dirty pages
- Clean pages

Políticas de Cache

1. Algoritmos de Substituição

- LRU (Least Recently Used)

```
class LRUCache<K,V> {  
    private final int capacity;  
    private LinkedHashMap<K,V> cache;  
  
    public LRUCache(int capacity) {  
        this.capacity = capacity;  
        this.cache = new LinkedHashMap<K,V>(capacity, 0.75f, true)  
    {  
        protected boolean removeEldestEntry(Map.Entry<K,V>  
eldest) {  
            return size() > capacity;  
        }  
    };  
}  
}
```

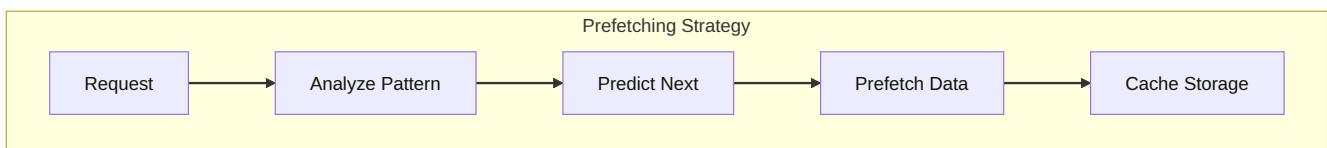
- Clock Algorithm

```
class ClockCache {  
    private Page[] buffer;  
    private int hand = 0;  
  
    public Page findVictim() {  
        while (true) {  
            if (buffer[hand].referenceBit == 0) {  
                return buffer[hand];  
            }  
            buffer[hand].referenceBit = 0;  
            hand = (hand + 1) % buffer.length;  
        }  
    }  
}
```

```
    }  
}  
}
```

2. Estratégias de Prefetching

- Sequential prefetch
- Index-based prefetch
- Pattern-based prefetch
- Adaptive prefetch



Otimizações

1. Cache-Conscious Design

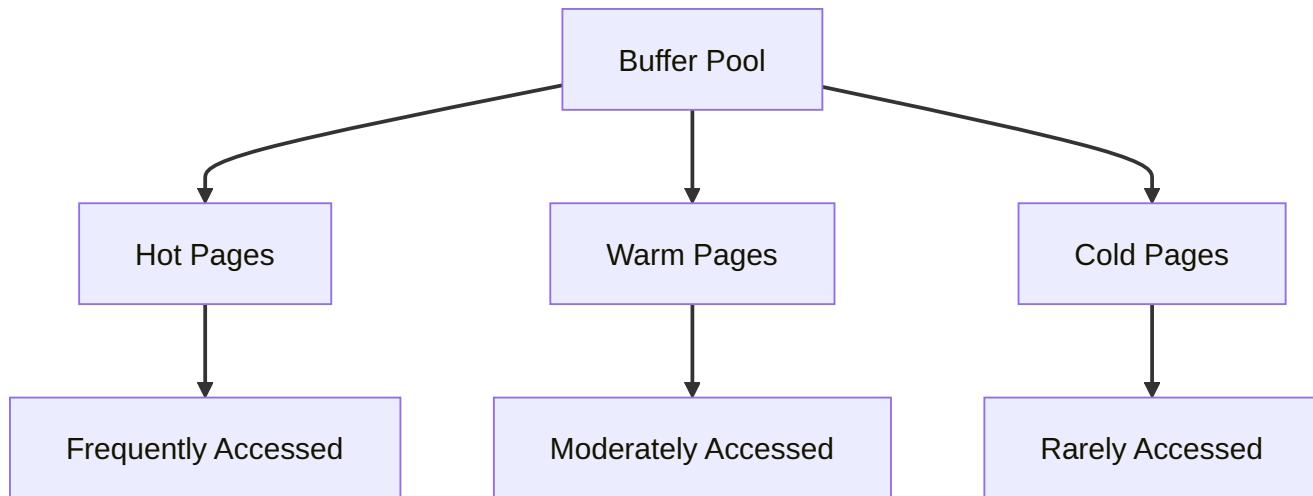
- Estruturas de Dados
 - Alinhamento de memória
 - Localidade espacial
 - Localidade temporal
 - Cache line padding

```
class CacheAlignedStruct {  
    private static final int CACHE_LINE = 64;  
  
    @Align(CACHE_LINE)  
    private long[] data;
```

```
private int pad; // Ensure alignment  
}
```

2. Técnicas Avançadas

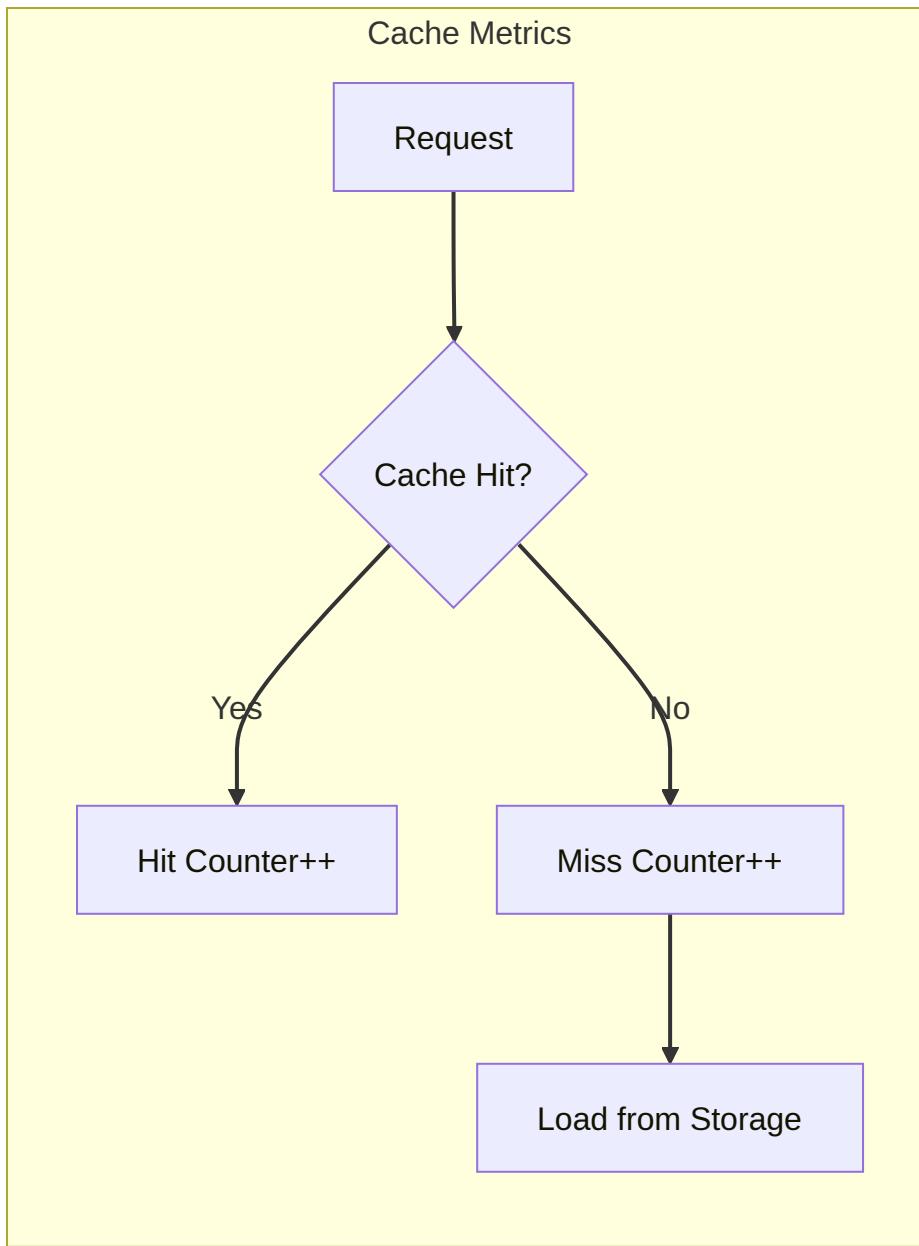
- Particionamento



Monitoramento e Diagnóstico

1. Métricas de Performance

- Indicadores Chave
 - Hit ratio
 - Miss ratio
 - Eviction rate
 - Response time



2. Ferramentas de Análise

- Cache profilers
- Memory analyzers
- Performance counters
- Monitoring tools

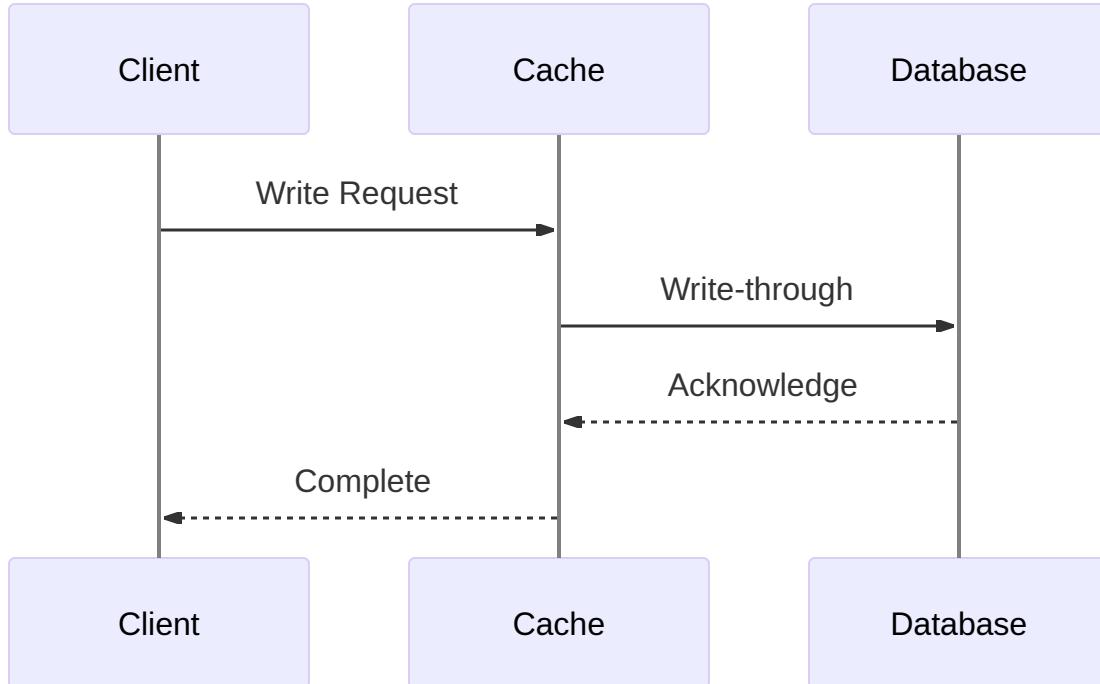
Implementação Prática

1. Cache Distribuído

```
interface DistributedCache {  
    void put(String key, Object value);  
    Object get(String key);  
    void invalidate(String key);  
    void clear();  
}  
  
class RedisCache implements DistributedCache {  
    private RedisClient client;  
  
    public void put(String key, Object value) {  
        client.set(key, serialize(value));  
    }  
  
    public Object get(String key) {  
        byte[] data = client.get(key);  
        return deserialize(data);  
    }  
}
```

2. Consistência e Sincronização

- Write-through vs Write-back
- Cache coherence
- Invalidation strategies
- Replication

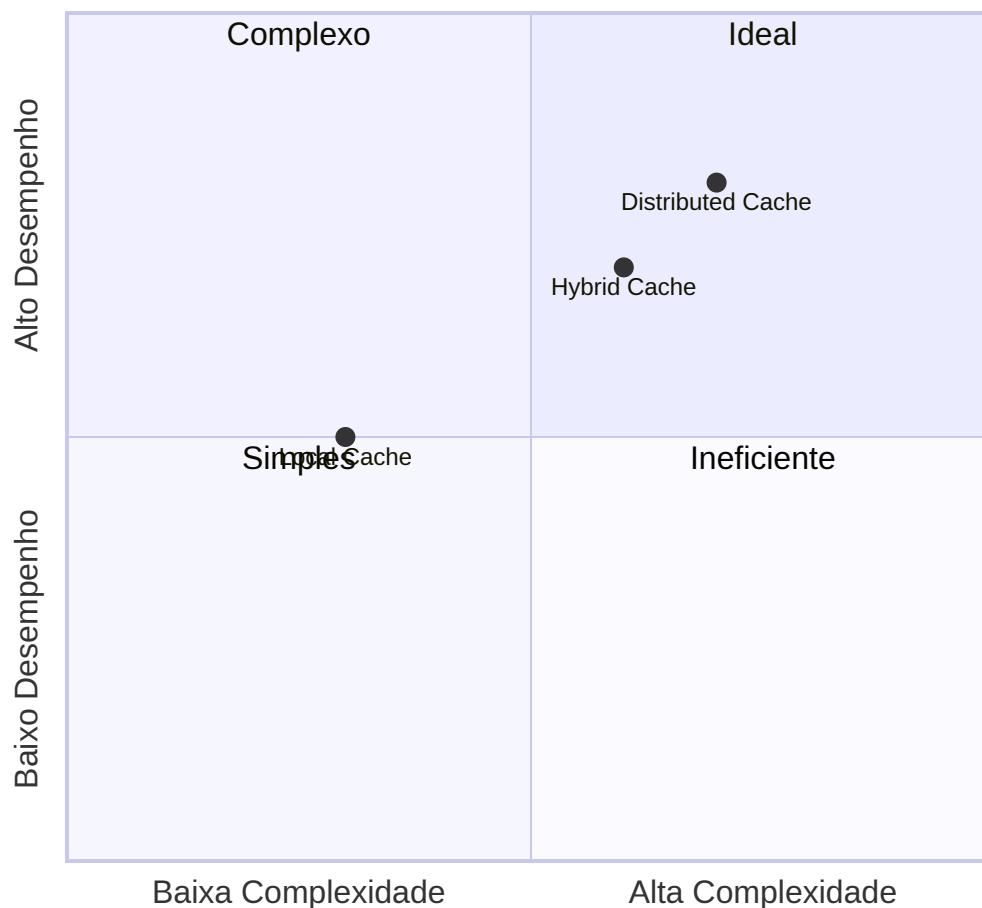


Considerações de Design

1. Trade-offs

- Tamanho vs Performance
- Consistência vs Latência
- Complexidade vs Flexibilidade
- Custo vs Benefício

Cache Design Trade-offs



2. Best Practices

- Cache warming
- Eviction policies
- Error handling
- Monitoring setup

Tendências e Inovações

1. Tecnologias Emergentes

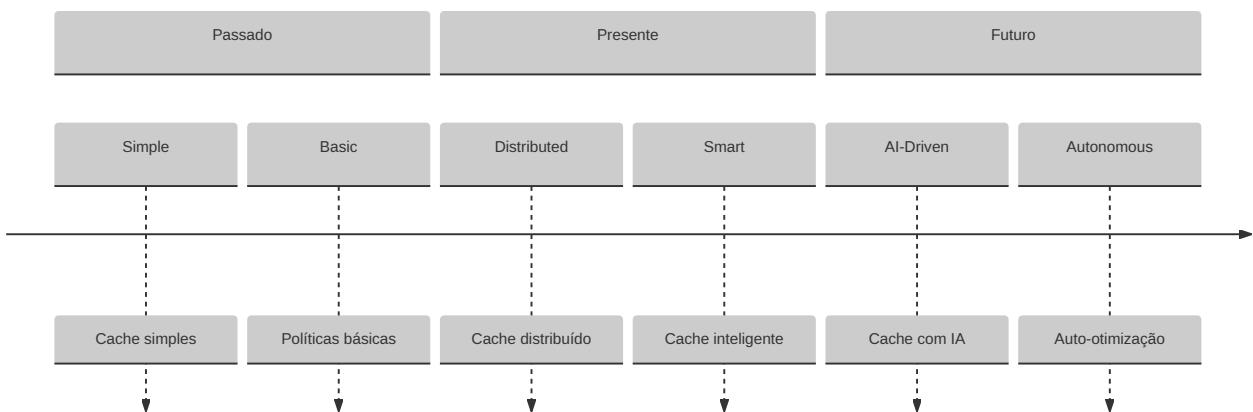
- AI/ML para cache prediction
- Hardware-assisted caching

- Persistent memory caching
- Smart prefetching

2. Futuras Direções

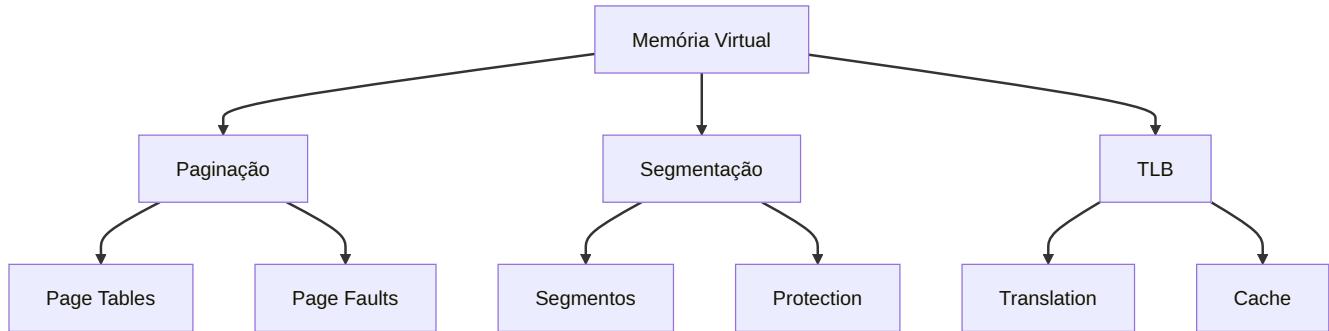
- Cache automation
- Self-tuning systems
- Intelligent prefetching
- Hybrid architectures

Evolução dos Sistemas de Cache



Memória Virtual

A memória virtual é um componente essencial dos sistemas modernos de banco de dados, fornecendo uma abstração entre a memória física e o espaço de endereçamento utilizado pelos processos.



Conceitos Fundamentais

1. Espaço de Endereçamento

- Endereçamento Virtual
 - Espaço linear
 - Independência de hardware
 - Isolamento de processos
 - Proteção de memória

2. Paginação

- Estrutura
 - Tamanho de página
 - Page frames
 - Page tables
 - Page directory

```

class PageTable {
    private static final int PAGE_SIZE = 4096;
    private PageEntry[] entries;

    class PageEntry {
        long physicalAddress;
        boolean present;
        boolean dirty;
        boolean referenced;
        int protection;
    }
}

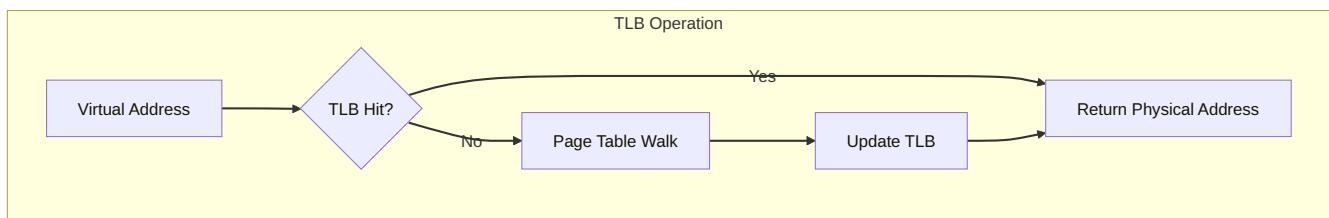
```

Mecanismos de Tradução

1. TLB (Translation Lookaside Buffer)

- Características

- Cache de traduções
- Hit rate
- Miss penalty
- Flush operations



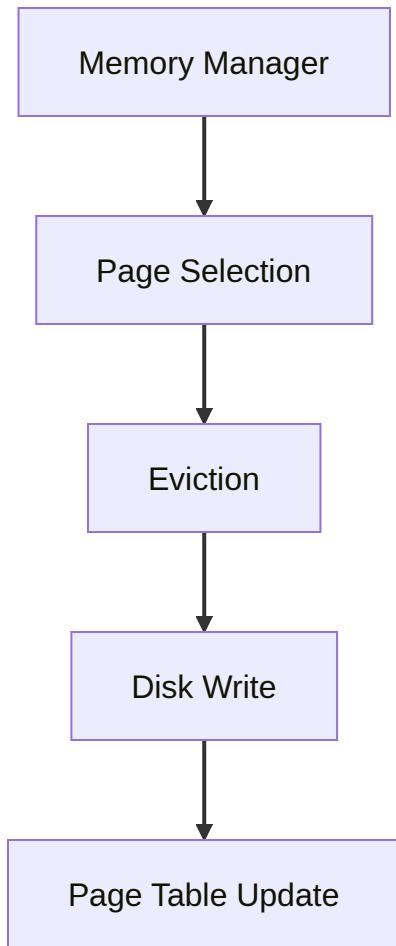
2. Page Fault Handling

```
class PageFaultHandler {  
    void handlePageFault(long virtualAddress) {  
        // 1. Localizar página no disco  
        Page page = findPageOnDisk(virtualAddress);  
  
        // 2. Encontrar frame livre  
        PhysicalFrame frame = findFreeFrame();  
  
        // 3. Carregar página  
        loadPage(page, frame);  
  
        // 4. Atualizar page table  
        updatePageTable(virtualAddress, frame);  
    }  
}
```

Otimizações

1. Técnicas de Gerenciamento

- Swapping
 - Políticas de substituição
 - Priorização de páginas
 - Working set
 - Thrashing prevention



2. Performance Tuning

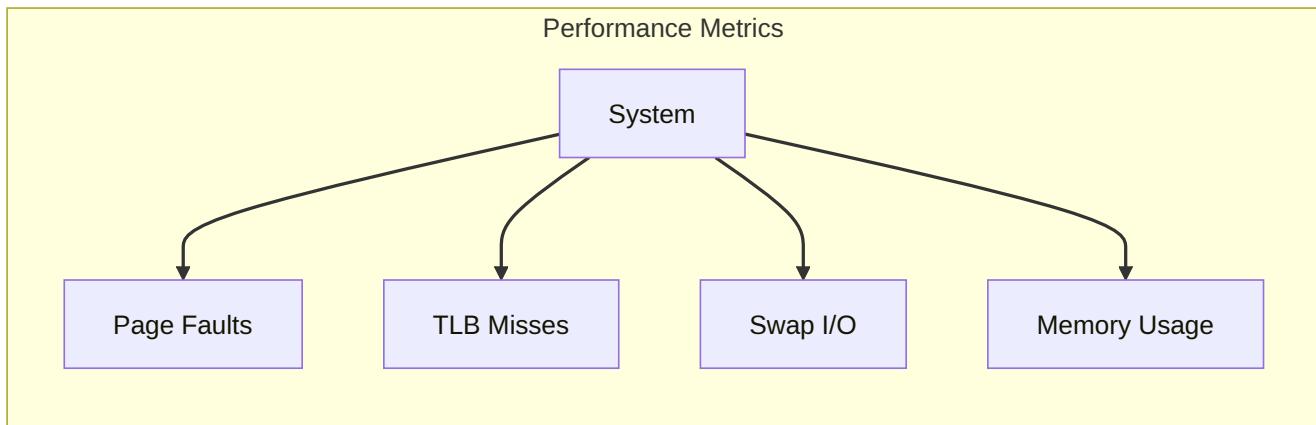
- Estratégias
 - Page size optimization
 - TLB coverage
 - Huge pages
 - Transparent huge pages

Monitoramento

1. Métricas Importantes

- Indicadores

- Page fault rate
- TLB miss rate
- Swap usage
- Memory pressure



2. Ferramentas de Análise

```

class MemoryMonitor {
    private MetricsCollector collector;

    public MemoryStats getStats() {
        return new MemoryStats(
            collector.getPageFaults(),
            collector.getTlbMisses(),
            collector.getSwapUsage(),
            collector.getMemoryPressure()
        );
    }
}
  
```

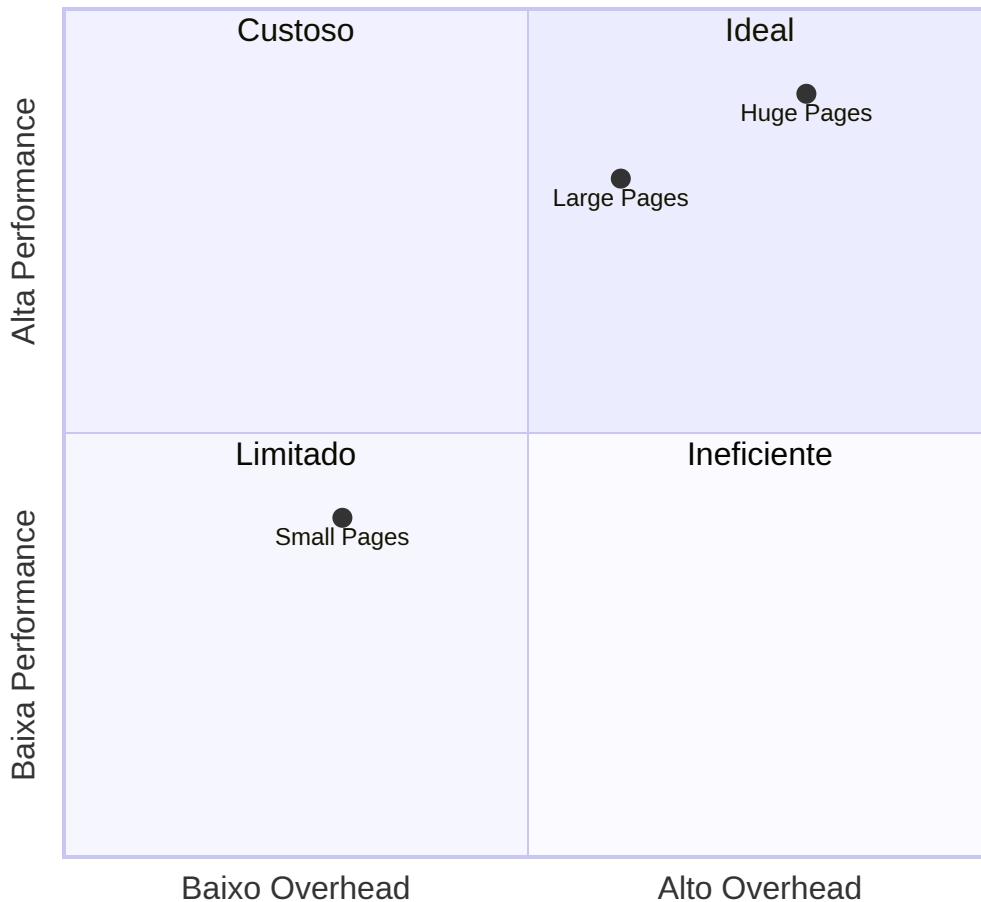
Considerações de Design

1. Trade-offs

- Balanceamento

- Tamanho de página
- TLB coverage
- Memory footprint
- I/O overhead

Memory Management Trade-offs



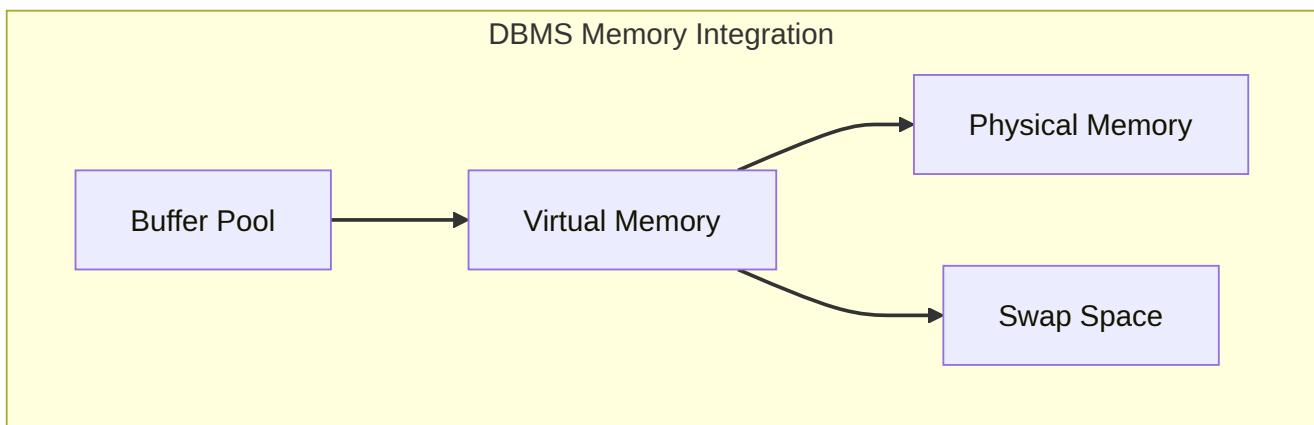
2. Best Practices

- Memory alignment
- Page coloring
- NUMA awareness
- Transparent huge pages

Integração com DBMS

1. Buffer Pool Management

- Coordenação
 - Page replacement
 - Buffer invalidation
 - Memory pressure handling
 - I/O scheduling



2. Otimizações Específicas

```
class DBMemoryManager {  
    private BufferPool bufferPool;  
    private VirtualMemoryManager vmManager;  
  
    public void optimizeMemory() {  
        // Ajusta buffer pool baseado em pressão de memória  
        long memoryPressure = vmManager.getMemoryPressure();  
        if (memoryPressure > threshold) {  
            bufferPool.shrink();  
        }  
    }  
}
```

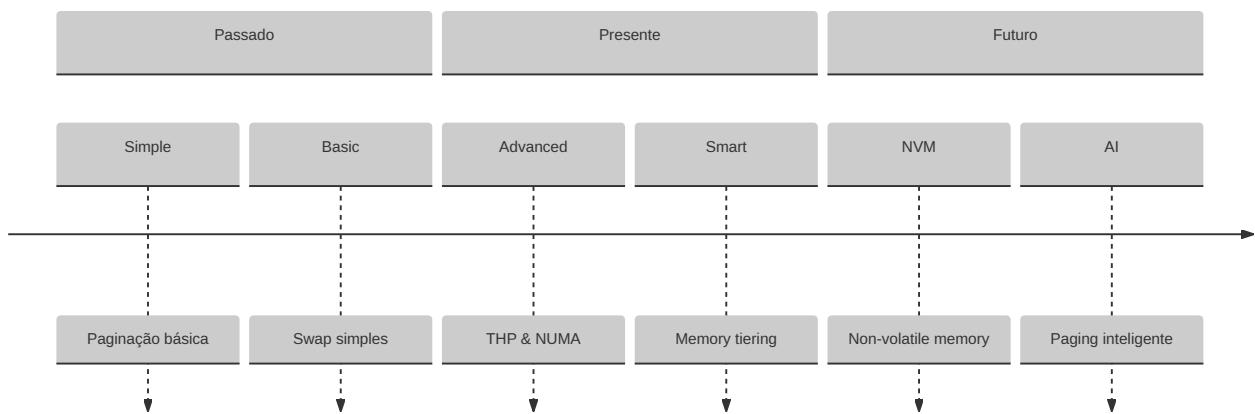
```
}
```

Tendências Futuras

1. Inovações

- Tecnologias Emergentes
 - Non-volatile memory
 - Disaggregated memory
 - Memory compression
 - Smart paging

Evolução da Memória Virtual



2. Direções Futuras

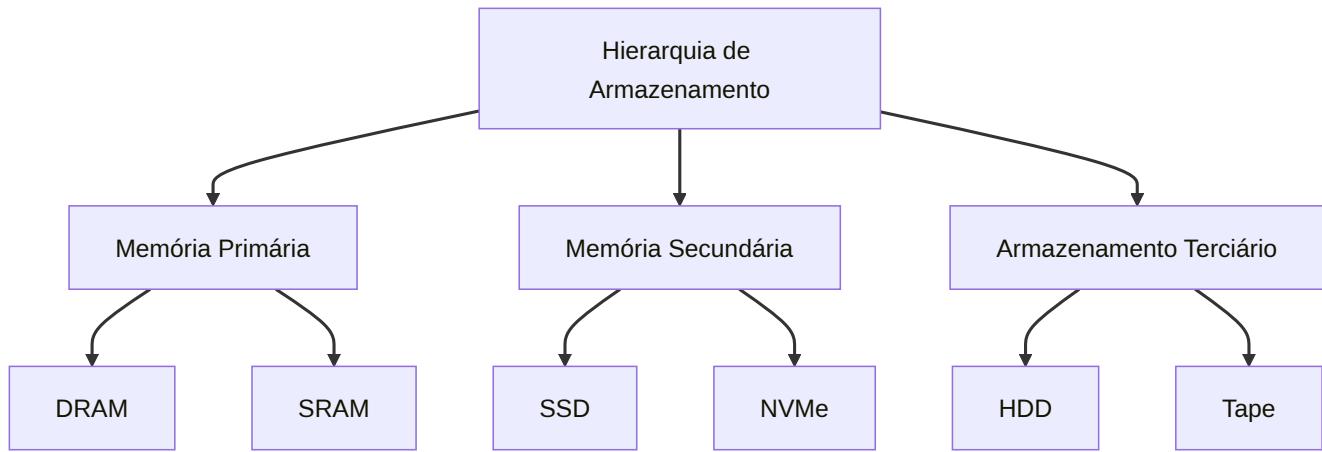
- AI/ML para predição de acesso
- Gerenciamento autônomo
- Integração com persistent memory
- Otimização dinâmica

Conclusão

A memória virtual continua sendo um componente crítico em sistemas de banco de dados modernos, evoluindo constantemente para atender às demandas crescentes de performance e eficiência.

Hierarquia de Armazenamento

A hierarquia de armazenamento é uma estrutura fundamental que organiza diferentes tecnologias de armazenamento baseadas em velocidade, custo e capacidade, impactando diretamente o desempenho dos sistemas de banco de dados.



Níveis de Armazenamento

1. Memória Primária

- Características
 - Acesso rápido
 - Volatilidade
 - Custo elevado
 - Capacidade limitada

2. Memória Secundária

- Tecnologias
 - SSDs
 - NVMe
 - Storage Class Memory

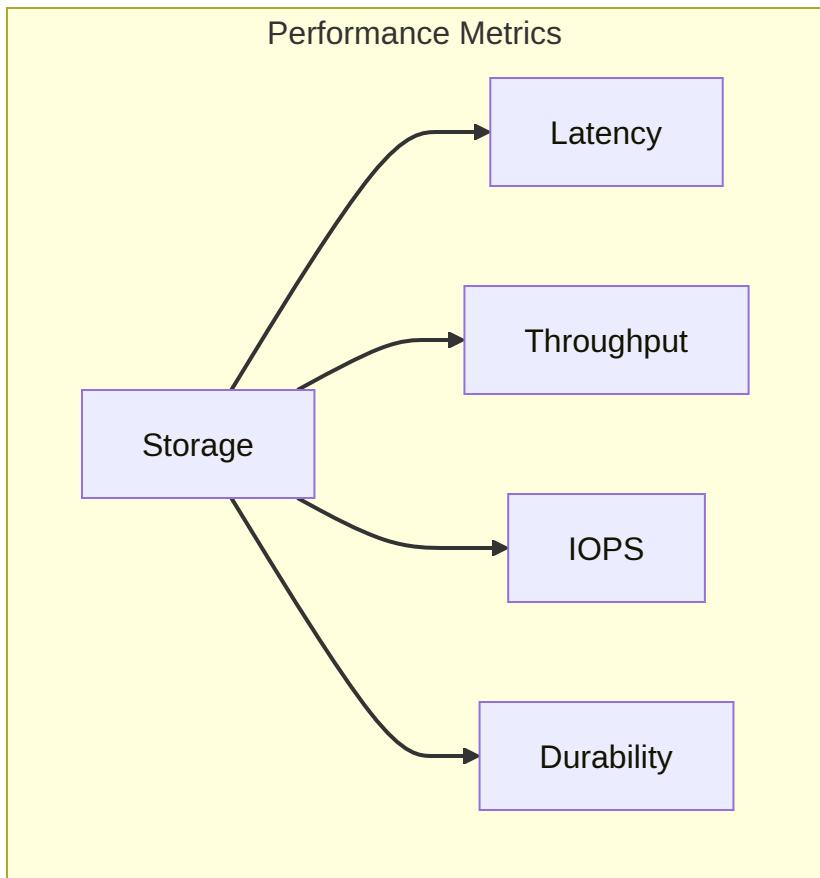
- Flash Arrays

```
class StorageDevice {  
    enum Type {  
        SSD, NVME, SCM, HDD  
    }  
  
    private final Type type;  
    private final long capacity;  
    private final int latency;  
    private final int throughput;  
}
```

Características de Performance

1. Métricas Principais

- Indicadores
 - Latência
 - Throughput
 - IOPS
 - Durabilidade



2. Trade-offs

```

class StorageManager {
    private Map<StorageTier, List<StorageDevice>> tiers;

    public void optimizePlacement(Data data) {
        StorageTier tier = selectOptimalTier(
            data.getAccessPattern(),
            data.getPriority(),
            data.getSize()
        );
        allocateToTier(data, tier);
    }
}

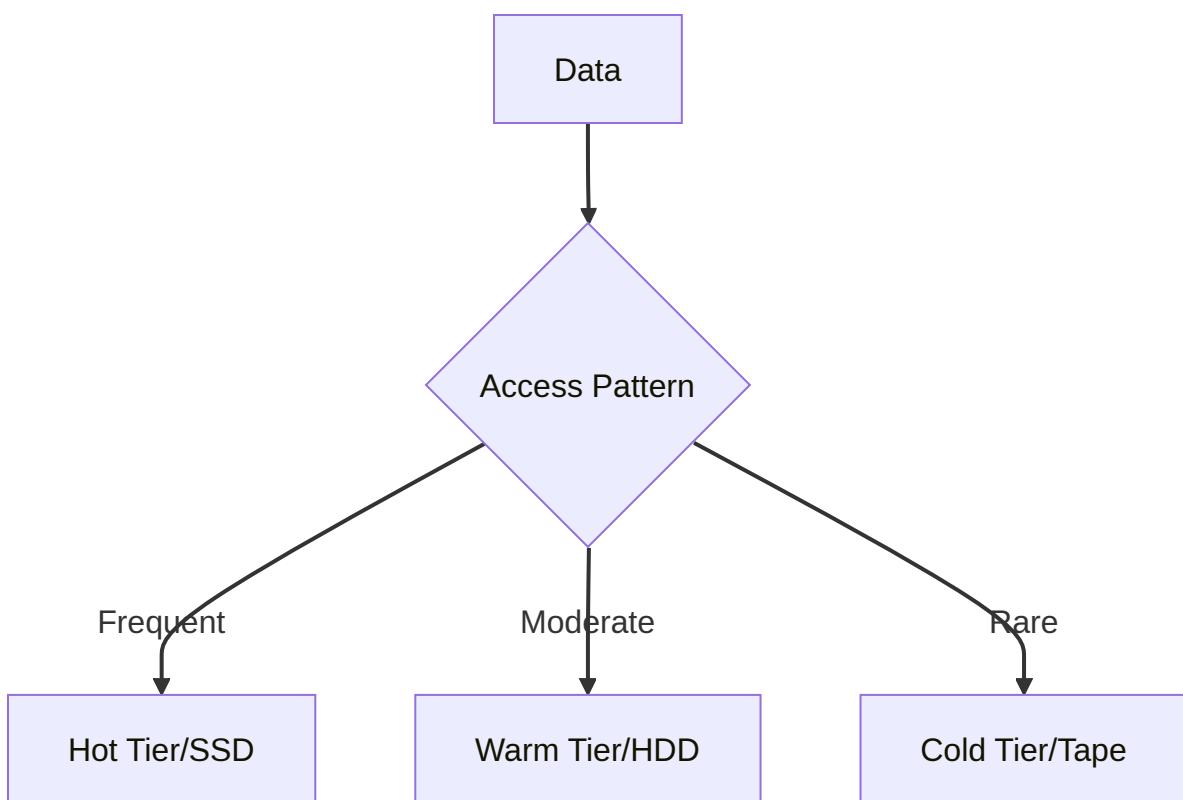
```

Estratégias de Gerenciamento

1. Tiered Storage

- Implementação

- Hot data
- Warm data
- Cold data
- Archive data



2. Caching Strategies

- Políticas

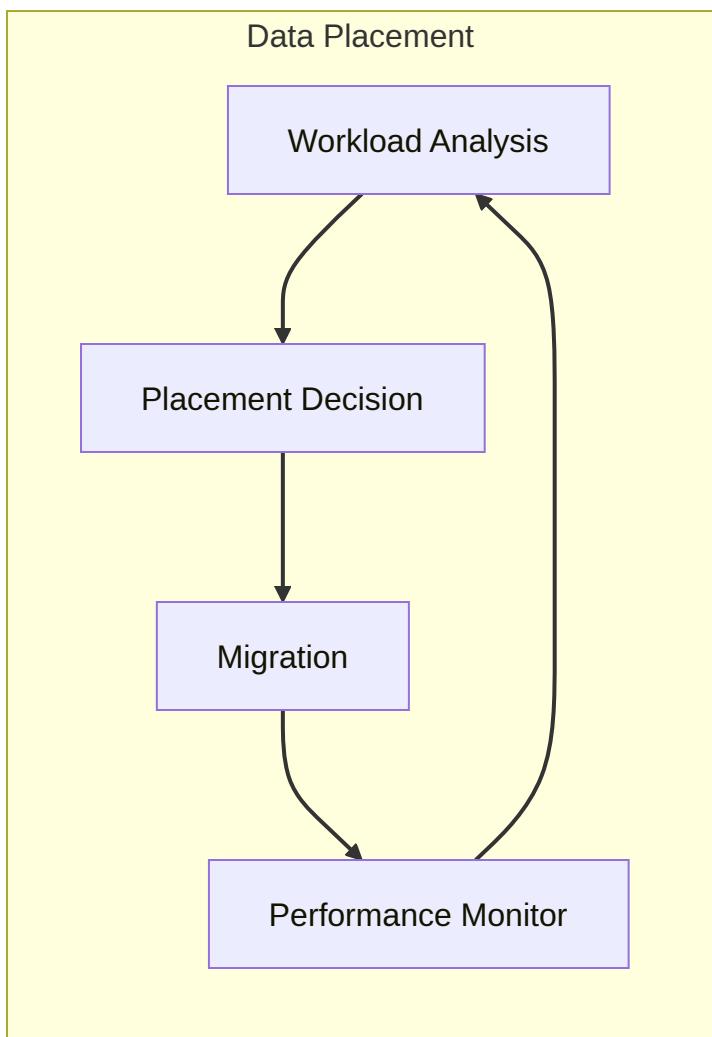
- Write-through
- Write-back
- Write-around

- Read-ahead

Otimizações

1. Data Placement

- Técnicas
 - Locality optimization
 - Access pattern analysis
 - Workload-based placement
 - Auto-tiering



2. I/O Optimization

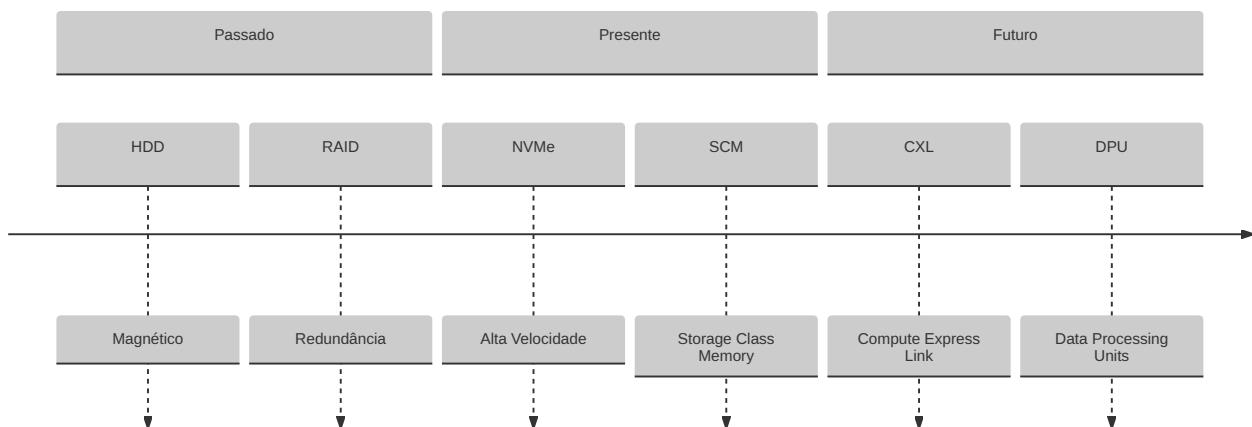
```
class IOOptimizer {  
    private IOScheduler scheduler;  
    private BufferManager buffer;  
  
    public void optimize() {  
        // Agrupa I/Os similares  
        List<IORequest> requests = scheduler.getRequests();  
        List<IORequest> optimized = mergeRequests(requests);  
  
        // Aplica write coalescing  
        buffer.coalesceWrites(optimized);  
    }  
}
```

Tecnologias Emergentes

1. Novas Arquiteturas

- Inovações
 - Persistent Memory
 - Storage Class Memory
 - Computational Storage
 - Disaggregated Storage

Evolução do Armazenamento



2. Tendências

- Direções
 - Inteligência artificial
 - Automação
 - Software-defined storage
 - Cloud-native storage

Considerações de Design

1. Arquitetura

- Aspectos
 - Escalabilidade
 - Disponibilidade
 - Consistência
 - Custo-benefício

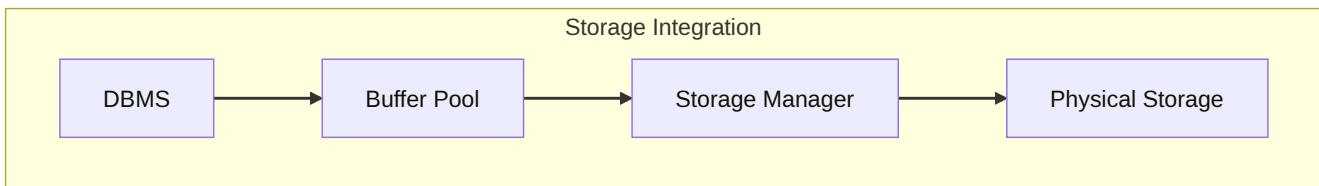
2. Best Practices

- Monitoramento contínuo
- Capacity planning
- Performance tuning
- Disaster recovery

Integração com DBMS

1. Buffer Management

- Estratégias
 - Page replacement
 - Prefetching
 - Write coalescing
 - I/O scheduling



2. Otimizações

```

class StorageOptimizer {
    private BufferPool bufferPool;
    private StorageManager storage;

    public void optimize() {
        // Ajusta buffer baseado em padrões de acesso
        AccessPattern pattern = analyzeAccess();
        adjustBufferSize(pattern);

        // Otimiza placement
        optimizePlacement(pattern);
    }
}

```

```
    }  
}
```

Conclusão

A hierarquia de armazenamento é um componente crítico que continua evoluindo com novas tecnologias e demandas, exigindo constante adaptação e otimização para maximizar o desempenho dos sistemas de banco de dados.

Modelagem de Dados

Visão Geral



Introdução à Modelagem de Dados

A modelagem de dados é um processo fundamental no desenvolvimento de sistemas de banco de dados, servindo como ponte entre os requisitos do negócio e a implementação técnica. Este processo envolve a criação de modelos que representam a estrutura, relacionamentos e restrições dos dados em diferentes níveis de abstração.

Objetivos da Modelagem

1. Representação da Realidade

- Capturar requisitos do negócio
- Mapear entidades e relacionamentos
- Definir regras e restrições

2. Qualidade dos Dados

- Garantir integridade
- Evitar redundância
- Manter consistência

3. Eficiência Operacional

- Otimizar consultas
- Facilitar manutenção
- Permitir escalabilidade

Níveis de Abstração

1. Nível Conceitual

- Foco no domínio do negócio
- Independente de tecnologia
- Diagrama Entidade-Relacionamento
- Visão de alto nível

2. Nível Lógico

- Estruturas de dados normalizadas
- Independente do SGBD
- Modelo Relacional
- Definição de chaves e relacionamentos

3. Nível Físico

- Específico para o SGBD
- Otimizações de performance
- Estruturas de armazenamento
- Índices e partições

Processo de Modelagem



Etapas do Processo

1. Levantamento de Requisitos

- Entrevistas com stakeholders
- Análise de documentação
- Identificação de regras de negócio

2. Análise de Dados

- Identificação de entidades
- Mapeamento de relacionamentos
- Definição de atributos

3. Desenvolvimento dos Modelos

- Criação do modelo conceitual
- Transformação para modelo lógico
- Refinamento do modelo físico

4. Validação e Refinamento

- Revisão com stakeholders
- Testes de consistência
- Ajustes de performance

Considerações de Design

1. Flexibilidade

- Adaptabilidade a mudanças
- Extensibilidade do modelo
- Reutilização de estruturas

2. Performance

- Otimização de consultas

- Estratégias de indexação
- Particionamento de dados

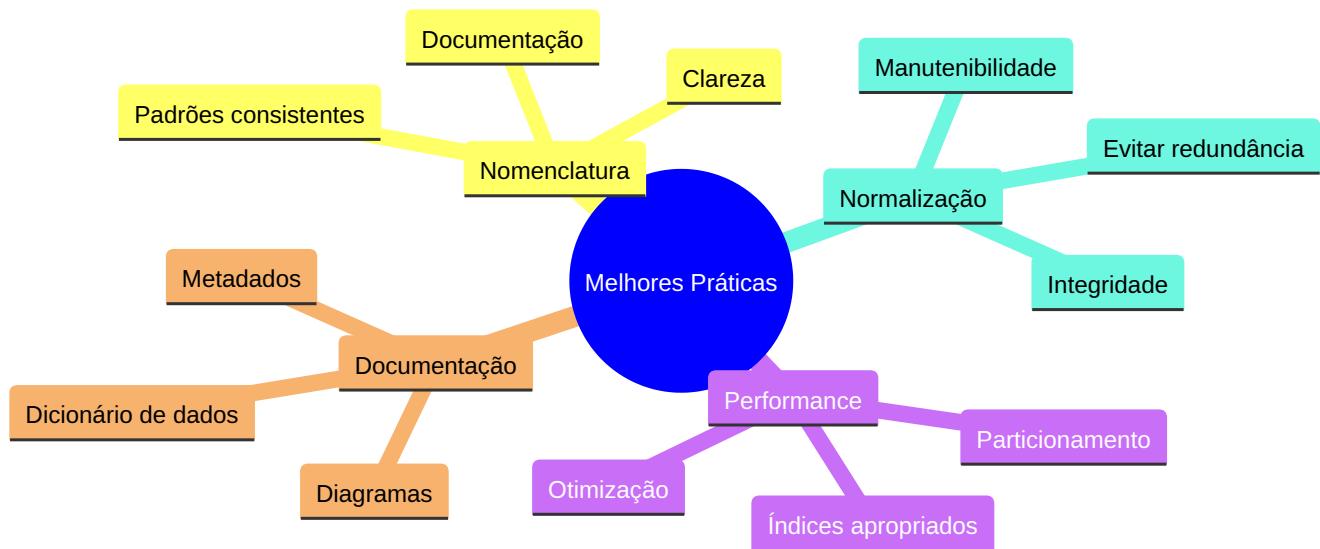
3. Integridade

- Consistência dos dados
- Regras de negócio
- Restrições de integridade

4. Segurança

- Controle de acesso
- Auditoria
- Proteção de dados

Melhores Práticas



1. Padronização

- Convenções de nomenclatura
- Documentação consistente

- Processos padronizados

2. Modularização

- Decomposição adequada
- Reutilização de estruturas
- Manutenibilidade

3. Validação

- Testes de integridade
- Verificação de requisitos
- Revisão por pares

Ferramentas e Tecnologias

1. Ferramentas CASE

- Modelagem visual
- Geração de código
- Documentação automática

2. Sistemas de Versionamento

- Controle de versão
- Colaboração em equipe
- Rastreabilidade

3. Frameworks de Modelagem

- Metodologias estabelecidas

- Padrões de indústria
- Melhores práticas

Conclusão

A modelagem de dados é uma disciplina fundamental que requer um equilíbrio entre teoria e prática. O sucesso de um projeto de banco de dados depende diretamente da qualidade de sua modelagem, que deve ser:

- Precisa na representação do negócio
- Eficiente em termos de performance
- Flexível para acomodar mudanças
- Manutenível a longo prazo

A combinação de boas práticas, ferramentas adequadas e uma metodologia sólida é essencial para criar modelos de dados que atendam às necessidades do presente e sejam adaptáveis às demandas futuras.

Modelagem Conceitual

Diagramas Entidade-Relacionamento (ER)

Os diagramas Entidade-Relacionamento (ER) são uma ferramenta fundamental para modelagem conceitual de dados. Eles descrevem as relações entre diferentes entidades em um domínio específico de conhecimento.

Componentes Básicos

1. Entidades

Uma entidade representa um objeto ou conceito do mundo real. Por convenção, os nomes das entidades são escritos em maiúsculas e no singular.

CLIENTE			
string	nome		
string	cpf	PK	Identificador único
string	email	UK	
string	telefone		

2. Relacionamentos

Os relacionamentos descrevem como as entidades se conectam entre si. A cardinalidade indica quantas instâncias de uma entidade podem se relacionar com outra.

Tipos de Cardinalidade:

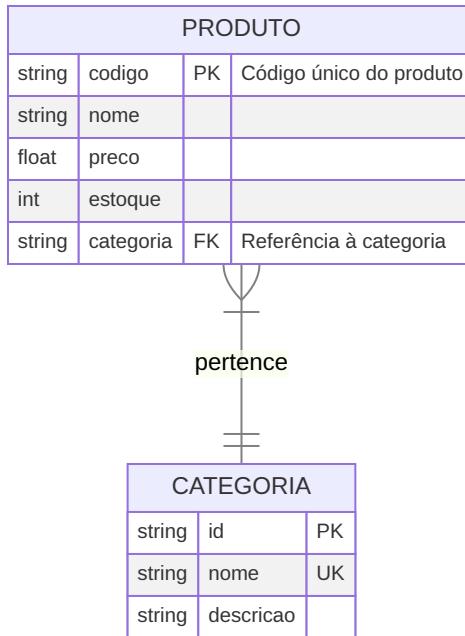
- ||--||: Exatamente um para um
- ||--o{: Um para muitos (zero ou mais)}
- ||--|{: Um para muitos (pelo menos um)}
- }o--o{: Muitos para muitos (zero ou mais)}

3. Atributos

Os atributos são características que descrevem uma entidade.

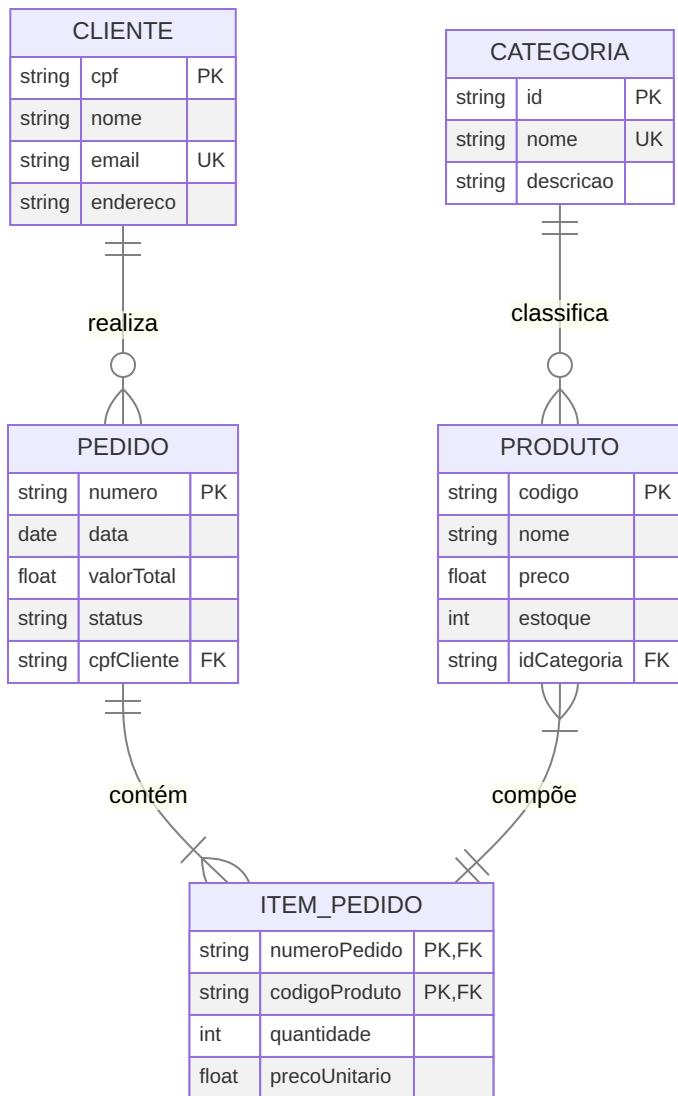
Tipos de Atributos:

- PK: Chave Primária
 - FK: Chave Estrangeira
 - UK: Chave Única



Exemplo Completo de Modelagem

Vamos modelar um sistema de e-commerce:



Boas Práticas

1. Nomenclatura

- Use nomes significativos
- Mantenha consistência
- Evite abreviações ambíguas

2. Cardinalidade

- Defina claramente as restrições

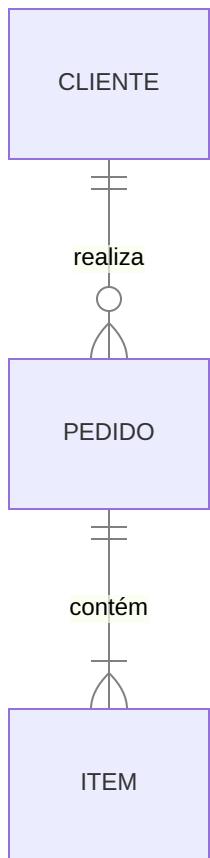
- Considere casos extremos
- Documente as regras de negócio

3. Atributos

- Identifique chaves primárias
- Estabeleça chaves estrangeiras
- Defina atributos obrigatórios

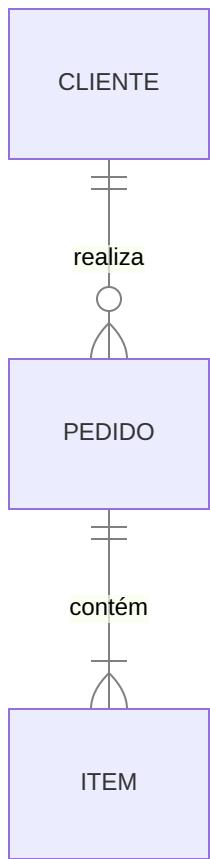
Direções do Diagrama

Os diagramas ER podem ser orientados em diferentes direções:



Estilização

É possível personalizar a aparência dos diagramas:



Conclusão

A modelagem ER é uma técnica poderosa para:

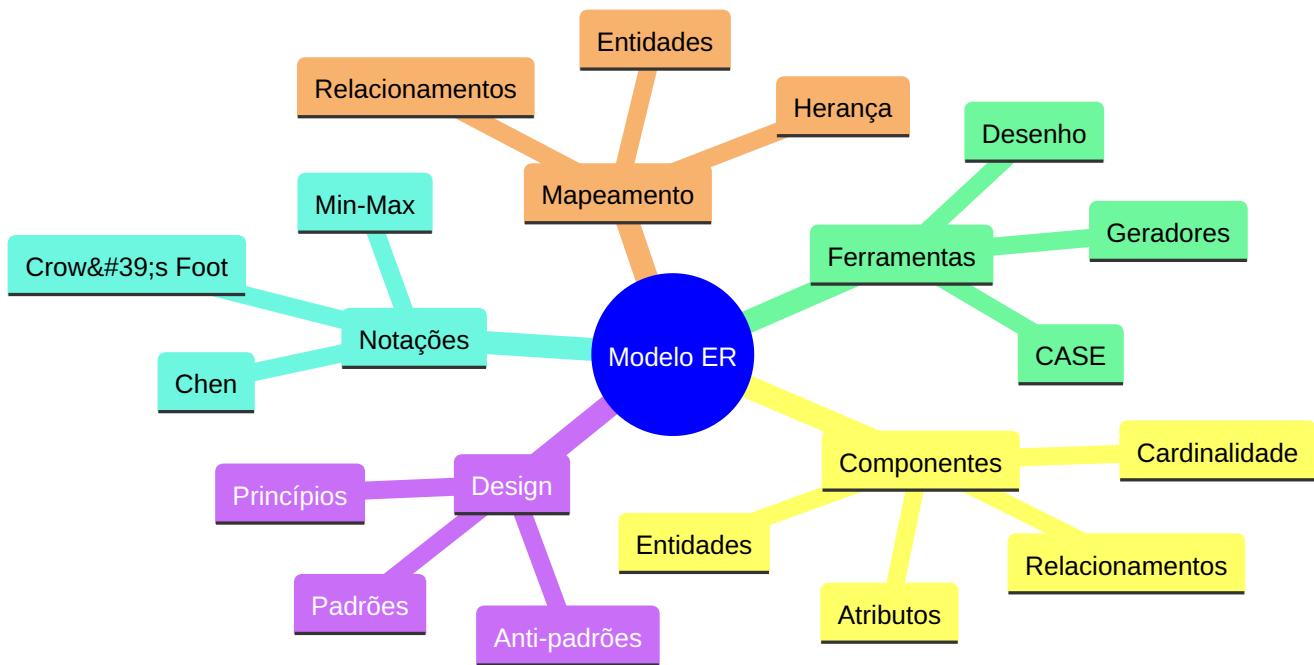
- Visualizar estruturas de dados
- Documentar requisitos
- Comunicar com stakeholders
- Planejar implementações

O uso de diagramas Mermaid torna a criação e manutenção desses modelos mais eficiente e integrada com documentação moderna.

Diagramas Entidade-Relacionamento (ER)

O modelo Entidade-Relacionamento (ER) é uma ferramenta fundamental para modelagem conceitual de dados, permitindo representar a estrutura lógica de um banco de dados de forma visual e intuitiva.

Visão Geral



Importância na Modelagem de Dados

O modelo ER serve como ponte entre os requisitos do negócio e a implementação técnica, oferecendo:

1. Comunicação Efetiva

- Facilita o diálogo entre stakeholders
- Representa visualmente conceitos complexos
- Documenta decisões de design

2. Abstração de Dados

- Foco na estrutura conceitual
- Independência de implementação
- Visão de alto nível do sistema

3. Base para Implementação

- Guia para modelo relacional
- Fundamento para design físico
- Referência para validação

Evolução Histórica

Princípios Fundamentais

1. Abstração

- Foco nos aspectos essenciais
- Omissão de detalhes técnicos
- Representação clara do domínio

2. Modularidade

- Decomposição em componentes
- Relacionamentos bem definidos
- Reutilização de padrões

3. Formalismo

- Regras claras de construção
- Semântica bem definida

- Consistência na representação

Benefícios e Limitações

Benefícios

1. Clareza Conceitual

- Fácil compreensão
- Representação intuitiva
- Documentação efetiva

2. Flexibilidade

- Adaptável a diferentes domínios
- Suporte a múltiplas notações
- Extensível para novos conceitos

3. Padronização

- Linguagem comum
- Práticas estabelecidas
- Ferramentas maduras

Limitações

1. Complexidade

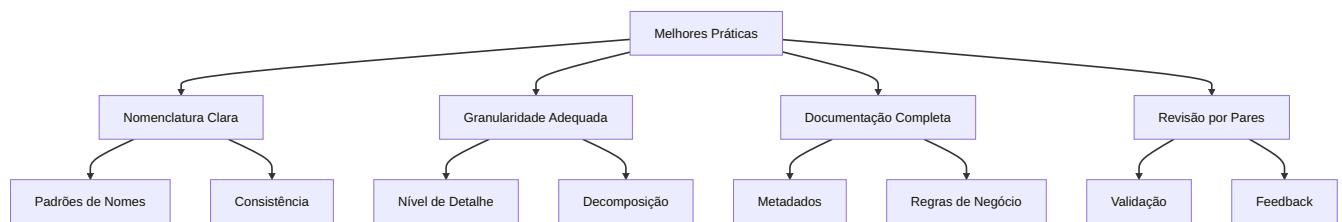
- Diagramas podem ficar sobrecarregados
- Necessidade de decomposição
- Curva de aprendizado inicial

2. Abstração vs. Detalhe

- Equilíbrio entre níveis

- Decisões de granularidade
- Compromissos de design

Melhores Práticas



1. Nomenclatura

- Use nomes significativos
- Mantenha consistência
- Siga convenções estabelecidas

2. Granularidade

- Defina nível apropriado de detalhe
- Decomponha modelos complexos
- Mantenha equilíbrio na abstração

3. Documentação

- Inclua metadados relevantes
- Documente regras de negócio
- Mantenha histórico de decisões

Próximos Passos

Para aprofundar seu conhecimento em modelagem ER, explore:

1. Conceitos Básicos

- Entidades e seus tipos
- Relacionamentos e cardinalidade
- Atributos e suas características

2. Restrições e Regras

- Chaves e identificadores
- Restrições de participação
- Regras de integridade

3. Notações e Ferramentas

- Diferentes estilos de notação
- Ferramentas de modelagem
- Técnicas de documentação

4. Design e Implementação

- Padrões de modelagem
- Mapeamento para modelo relacional
- Otimizações e refinamentos

Conclusão

O modelo ER continua sendo uma ferramenta essencial para modelagem de dados, oferecendo:

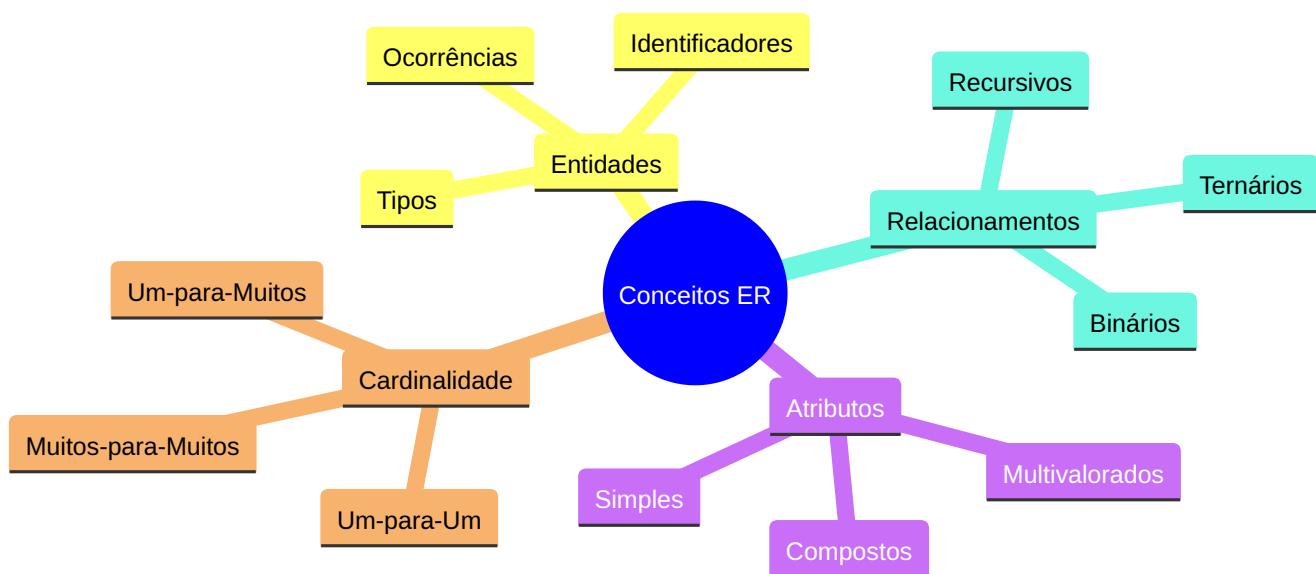
- Base sólida para design de banco de dados
- Comunicação efetiva entre stakeholders
- Documentação clara e manutenível
- Fundamento para implementação técnica

A compreensão profunda dos conceitos ER e suas aplicações é fundamental para qualquer profissional de banco de dados.

Conceitos Fundamentais do Modelo ER

Os conceitos fundamentais do Modelo Entidade-Relacionamento (ER) formam a base para a modelagem conceitual de dados. Este capítulo explora os elementos essenciais que compõem um diagrama ER.

Visão Geral dos Conceitos



Elementos Básicos

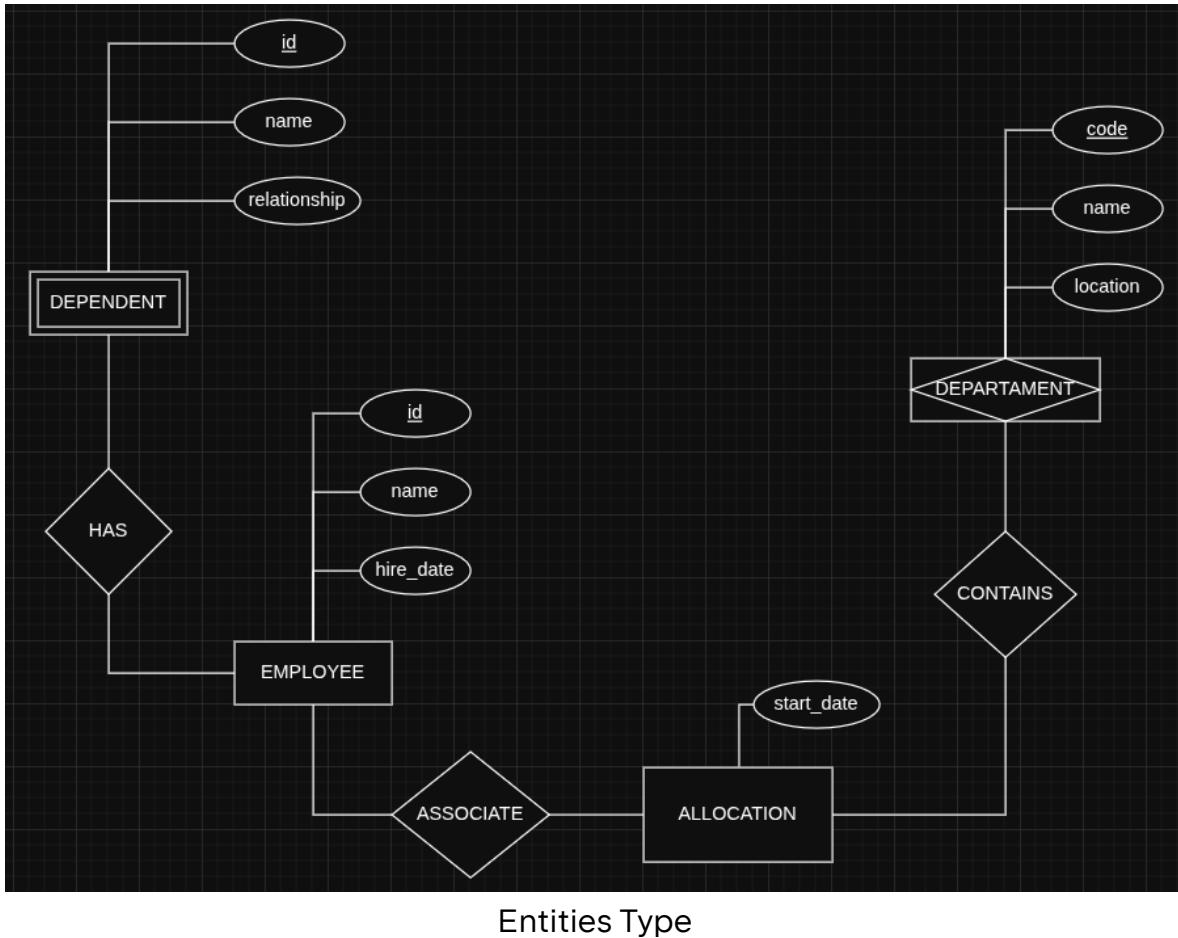
1. Entidades

Uma entidade representa um objeto ou conceito do mundo real que pode ser distintamente identificado.

Tipos de Entidades

- **Entidades Fortes:** Existem independentemente de outras entidades
- **Entidades Fracas:** Dependem de outras entidades para existir
- **Entidades Associativas:** Resultam da associação entre outras entidades
- **Exemplo:**

- EMPLOYEE (entidade forte)
- DEPENDENT (entidade fraca)
- DEPARTAMENT (entidade associativa)



Entities Type

2. Relacionamentos

Representam associações entre entidades, descrevendo como elas interagem entre si.

⚠ Nunca crie relacionamentos entre atributos

Sempre crie relacionamentos entre entidades!

Características dos Relacionamentos

- **Grau:** Número de entidades participantes

- **Papel:** Função de cada entidade no relacionamento

3. Atributos

Descrevem propriedades ou características das entidades e relacionamentos.

Classificação dos Atributos

1. Quanto à Estrutura

- Simples (atômicos)
- Compostos
- Multivalorados

2. Quanto à Função

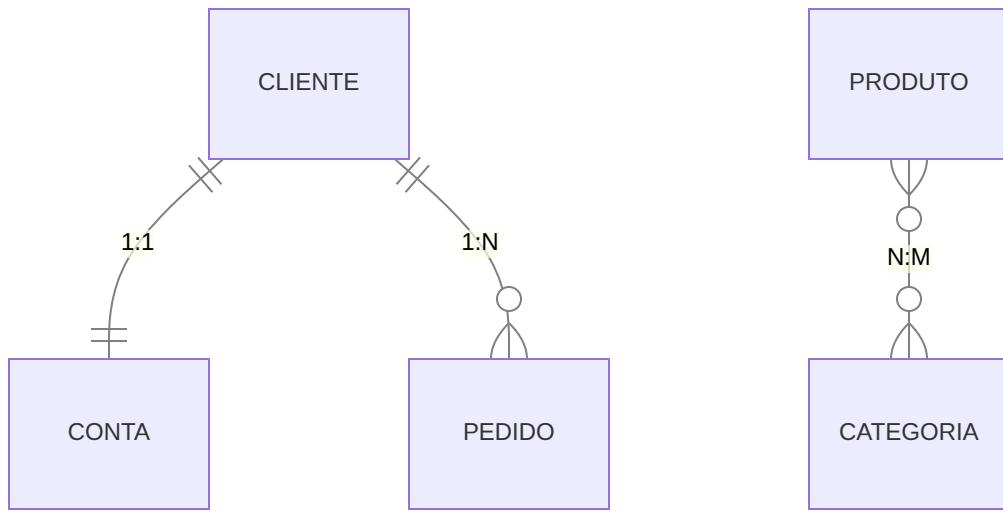
- Descritivos
- Identificadores (chaves)
- Referenciais

PESSOA			
string	cpf	PK	Identificador
string	nome		Simples
string	endereco		Composto
string[]	telefones		Multivalorado

4. Cardinalidade

Define o número de instâncias de uma entidade que podem estar relacionadas com instâncias de outra entidade.

Tipos de Cardinalidade



- **Um-para-Um (1:1)**

- Cada instância se relaciona com no máximo uma instância
- Exemplo: Pessoa ↔ CPF

- **Um-para-Muitos (1:N)**

- Uma instância se relaciona com várias instâncias
- Exemplo: Departamento ↔ Funcionários

- **Muitos-para-Muitos (N:M)**

- Várias instâncias se relacionam com várias instâncias
- Exemplo: Alunos ↔ Disciplinas

Regras e Restrições

1. Integridade Referencial

- Garante consistência entre relacionamentos
- Previne referências inválidas
- Mantém a coerência dos dados

2. Participação

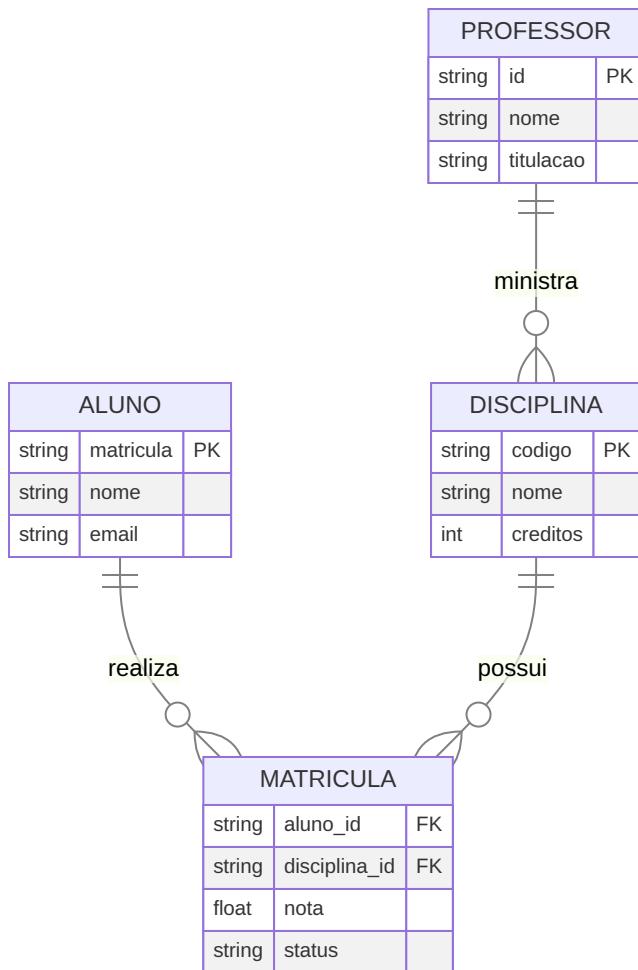
- **Total:** Todas as instâncias participam do relacionamento
- **Parcial:** Algumas instâncias podem não participar

3. Exclusividade

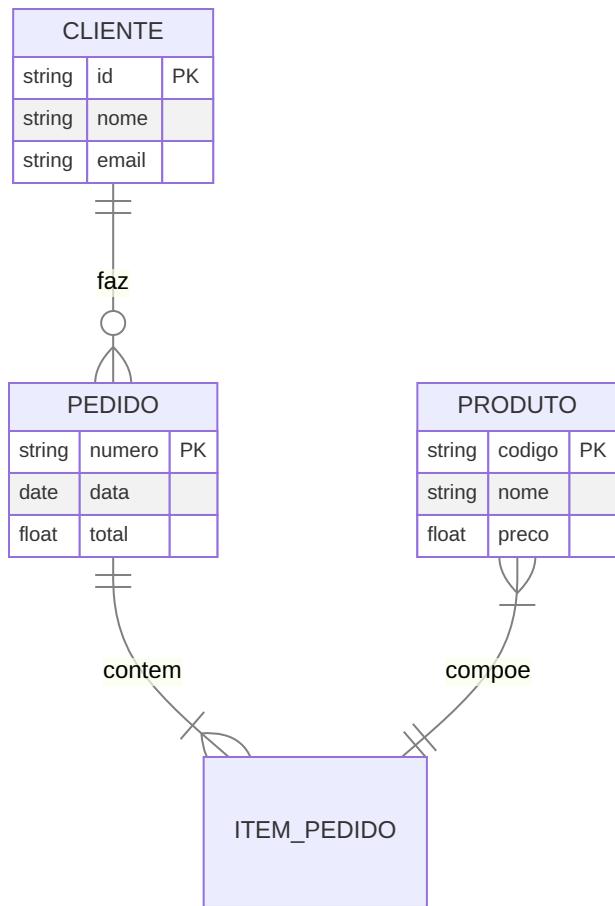
- Define se uma instância pode participar em múltiplos relacionamentos
- Estabelece restrições de unicidade

Exemplos Práticos

Sistema Acadêmico



Sistema de E-commerce



Considerações de Design

1. Normalização vs. Desnormalização

- Equilíbrio entre integridade e desempenho
- Decisões baseadas em requisitos

2. Granularidade

- Nível adequado de detalhamento
- Decomposição de entidades complexas

3. Manutenibilidade

- Facilidade de evolução

- Clareza na representação

Conclusão

Os conceitos fundamentais do Modelo ER são essenciais para:

- Compreensão da estrutura de dados
- Comunicação entre stakeholders
- Base para implementação física
- Documentação do sistema

A aplicação correta destes conceitos resulta em modelos:

- Claros e compreensíveis
- Precisos e consistentes
- Flexíveis e extensíveis
- Implementáveis e manutêveis

Entidades

Uma entidade é um objeto ou conceito do mundo real que pode ser identificado de forma única e sobre o qual desejamos armazenar informações.

Características das Entidades

1. Identificação Única

Cada entidade deve possuir um identificador único (chave primária) que a distingue das demais.

PRODUTO			
string	codigo	PK	Identificador único
string	nome		
float	preco		

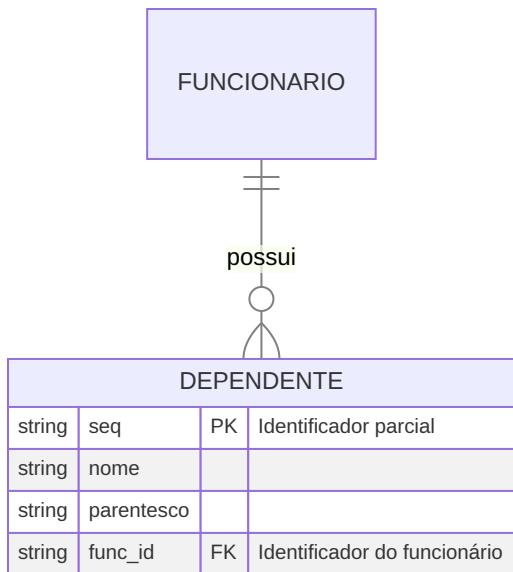
2. Tipos de Entidades

Entidades Fortes

- Existem independentemente de outras entidades
- Possuem identificador próprio
- Exemplo: CLIENTE, PRODUTO

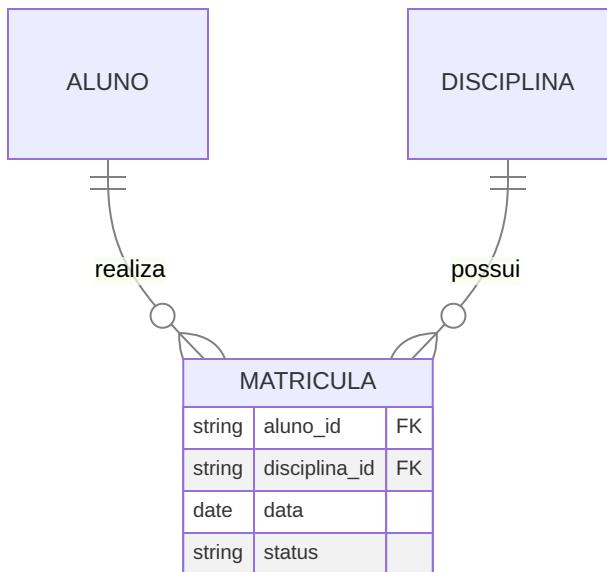
Entidades Fracas

- Dependem de outras entidades para existir
- Identificador parcial
- Exemplo: DEPENDENTE, ITEM_PEDIDO



Entidades Associativas

- Resultam do relacionamento entre outras entidades
- Também conhecidas como entidades de junção
- Exemplo: MATRICULA, INSCRICAO



3. Ocorrências (Instâncias)

Uma ocorrência é uma instância específica de uma entidade.

Exemplo para entidade PRODUTO:

- {codigo: "001", nome: "Laptop", preco: 3500.00}
- {codigo: "002", nome: "Mouse", preco: 89.90}

Boas Práticas

1. Nomenclatura

- Use substantivos no singular
- Evite abreviações
- Use maiúsculas para nomes de entidades
- Seja consistente com o padrão adotado

2. Identificação

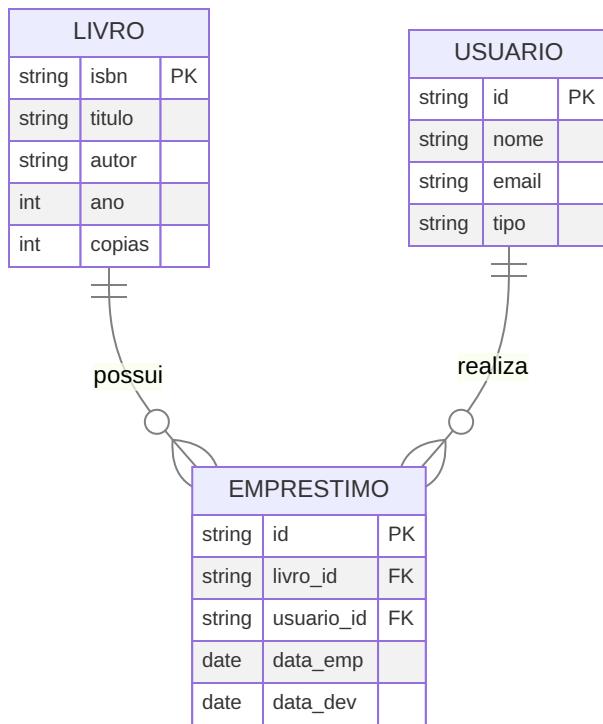
- Escolha identificadores estáveis
- Prefira chaves naturais quando possível
- Use chaves surrogate quando necessário

3. Granularidade

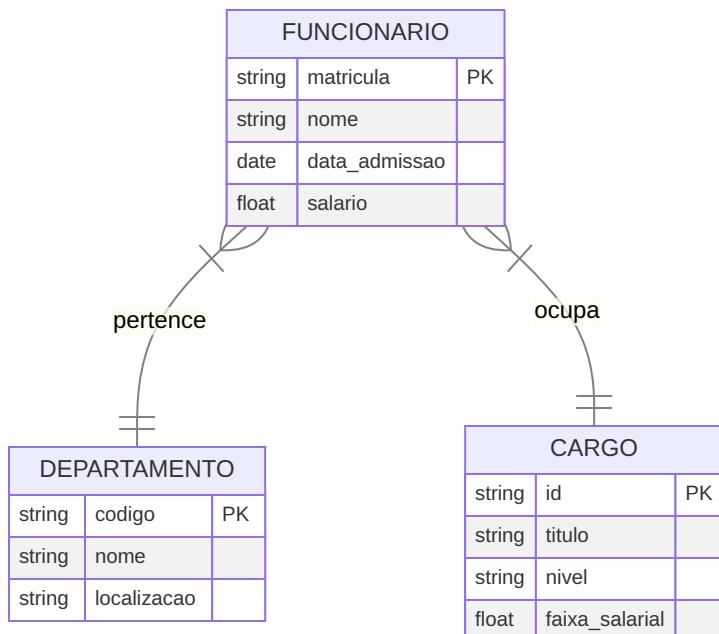
- Defina o nível adequado de abstração
- Evite entidades muito genéricas
- Evite entidades muito específicas

Exemplos Práticos

Sistema de Biblioteca



Sistema de RH



Considerações Importantes

1. Integridade

- Garanta que cada entidade tenha identificador único
- Mantenha a consistência dos dados
- Defina regras de validação

2. Relacionamentos

- Identifique corretamente as dependências
- Estabeleça cardinalidades apropriadas
- Considere o ciclo de vida das entidades

3. Evolução

- Planeje para mudanças futuras
- Documente decisões de design
- Mantenha o modelo atualizado

Conclusão

Entidades são fundamentais para:

- Organização dos dados
- Representação do domínio
- Base para implementação
- Comunicação entre stakeholders

A modelagem correta de entidades é crucial para:

- Integridade dos dados
- Eficiência do sistema
- Manutenibilidade

- Escalabilidade

Relacionamentos

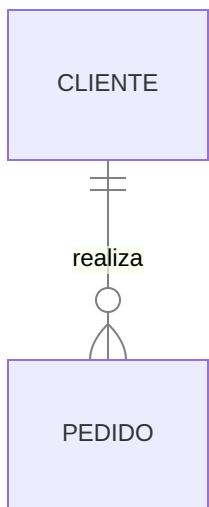
Um relacionamento representa uma associação entre duas ou mais entidades, descrevendo como elas interagem entre si no contexto do domínio.

Características dos Relacionamentos

1. Grau do Relacionamento

Relacionamento Binário

- Envolve duas entidades
- Tipo mais comum
- Exemplo: Cliente realiza Pedido

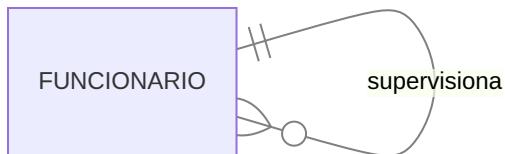


Relacionamento Ternário

- Envolve três entidades
- Usado em casos específicos
- Exemplo: Professor leciona Disciplina para Turma

Relacionamento Recursivo

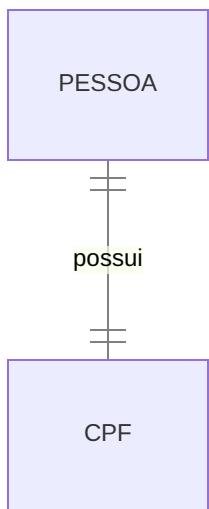
- Uma entidade se relaciona consigo mesma
- Exemplo: Funcionário supervisiona Funcionário



2. Cardinalidade

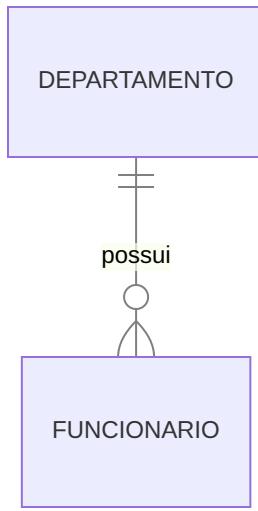
Um-para-Um (1:1)

- Cada instância se relaciona com no máximo uma outra
- Exemplo: Pessoa tem um CPF



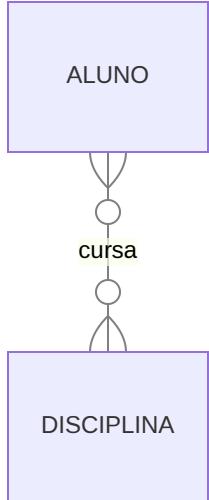
Um-para-Muitos (1:N)

- Uma instância se relaciona com várias outras
- Exemplo: Departamento tem vários Funcionários



Muitos-para-Muitos (N:M)

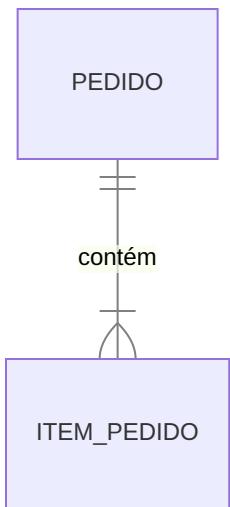
- Várias instâncias se relacionam com várias outras
- Exemplo: Aluno cursa várias Disciplinas



3. Participação

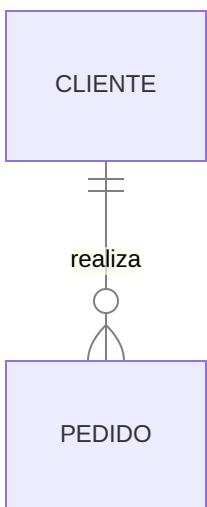
Participação Total

- Todas as instâncias da entidade participam do relacionamento
- Representada por linha dupla



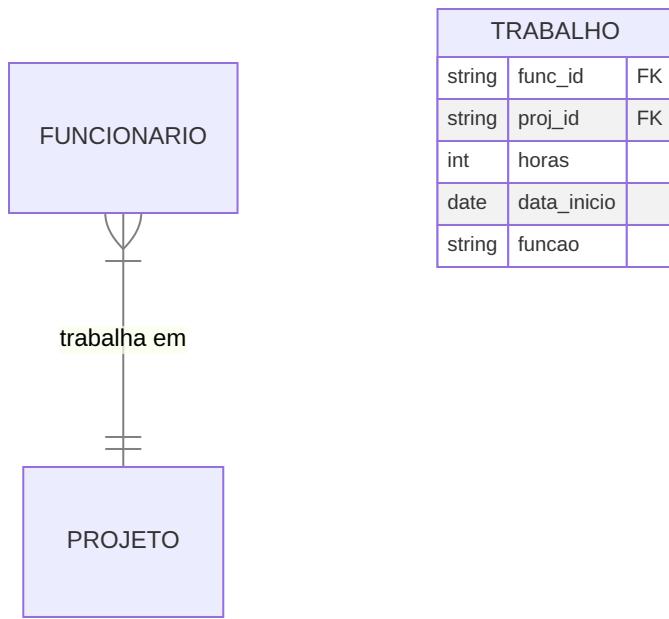
Participação Parcial

- Algumas instâncias podem não participar
- Representada por linha simples



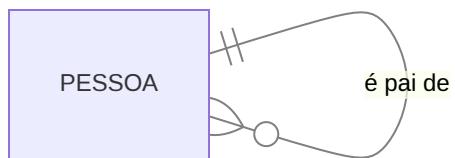
Atributos em Relacionamentos

Relacionamentos podem ter seus próprios atributos.



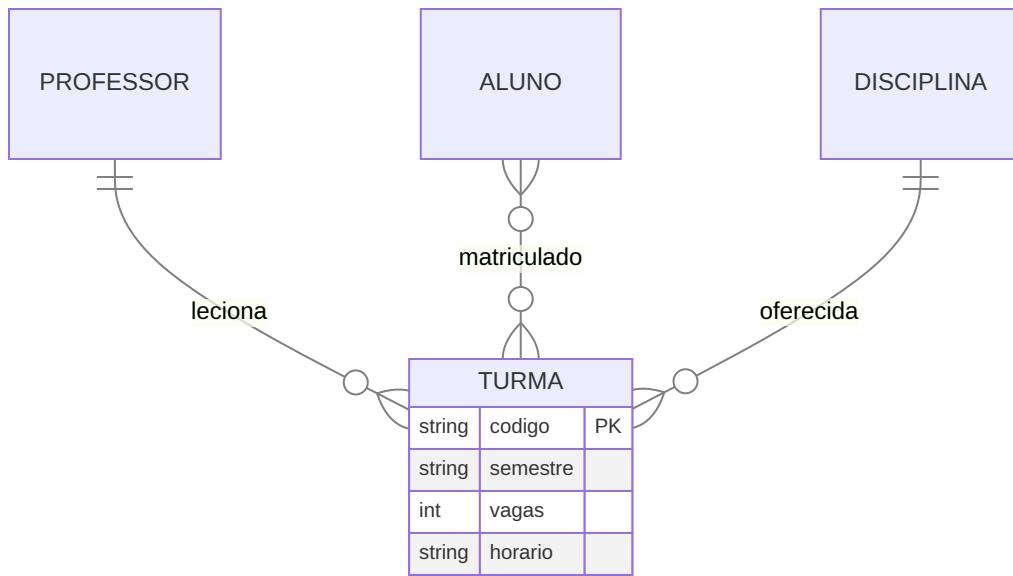
Papéis em Relacionamentos

Cada entidade desempenha um papel específico no relacionamento.

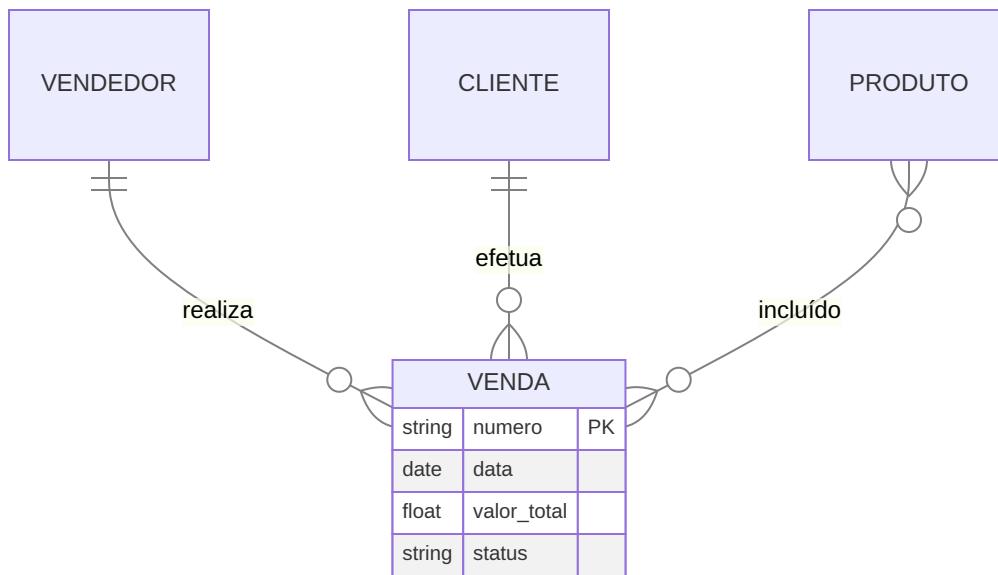


Exemplos Práticos

Sistema Acadêmico



Sistema de Vendas



Boas Práticas

1. Nomenclatura

- Use verbos para nomes de relacionamentos
- Seja claro e específico

- Mantenha consistência

2. Design

- Evite relacionamentos redundantes
- Minimize relacionamentos complexos
- Documente restrições importantes

3. Implementação

- Considere o impacto na performance
- Planeje índices adequados
- Mantenha a integridade referencial

Considerações de Modelagem

1. Normalização

- Balance normalização com performance
- Considere requisitos de consulta
- Avalie impacto nas operações

2. Restrições

- Defina regras de negócio
- Implemente validações
- Mantenha consistência

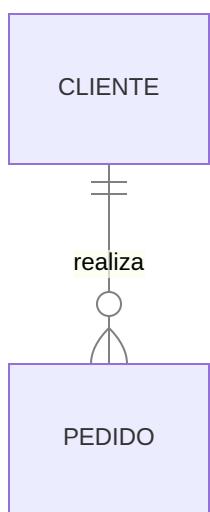
3. Evolução

- Planeje para mudanças

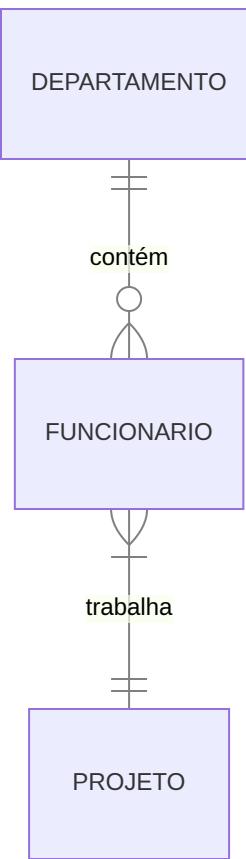
- Documente decisões
- Mantenha flexibilidade

Padrões Comuns

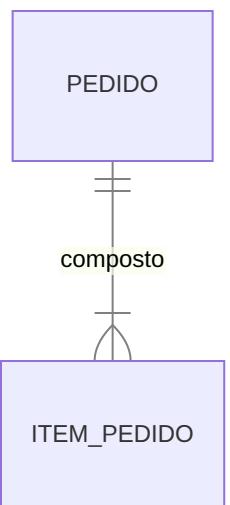
1. Associação Simples



2. Agregação



3. Composição



Conclusão

Relacionamentos são essenciais para:

- Estruturação dos dados
- Integridade do modelo
- Representação do negócio
- Implementação eficiente

A modelagem adequada de relacionamentos:

- Facilita a manutenção
- Melhora a performance
- Garante consistência
- Suporta evolução

Atributos

Atributos são propriedades ou características que descrevem entidades e relacionamentos em um modelo ER.

Tipos de Atributos

1. Quanto à Estrutura

Atributos Simples

- Valores atômicos, indivisíveis
- Exemplo: CPF, idade, email

CLIENTE		
string	cpf	PK
int	idade	
string	email	

Atributos Compostos

- Podem ser divididos em partes menores
- Exemplo: endereço (rua, número, cidade)

Atributos Multivalorados

- Podem ter múltiplos valores
- Exemplo: telefones, emails

PESSOA			
string	id	PK	
string	nome		
string[]	telefones		Múltiplos valores
string[]	emails		Múltiplos valores

2. Quanto à Função

Atributos Identificadores (Chaves)

- **Chave Primária (PK)**
 - Identifica unicamente cada instância
 - Não pode ser nulo ou duplicado

PRODUTO			
string	codigo	PK	Identificador único
string	nome		
float	preco		

- **Chave Estrangeira (FK)**
 - Referencia chave primária de outra entidade
 - Estabelece relacionamentos

PEDIDO			
string	numero	PK	
string	cliente_id	FK	Referência ao cliente
date	data		
float	valor		

- **Chave Única (UK)**
 - Valor único mas não primário

- Exemplo: email, matrícula

USUARIO			
string	id	PK	
string	email	UK	Único mas não primário
string	nome		

Atributos Descritivos

- Descrevem características
- Não são identificadores
- Exemplo: nome, descrição, data

LIVRO			
string	isbn	PK	
string	titulo		Descriutivo
string	autor		Descriutivo
int	paginas		Descriutivo

3. Quanto ao Valor

Atributos Obrigatórios

- Não podem ser nulos
- Essenciais para a entidade

FUNCIONARIO			
string	matricula	PK	Obrigatório
string	nome		Obrigatório
string	cargo		Obrigatório
string	telefone		Opcional

Atributos Opcionais

- Podem ser nulos
- Não essenciais

Atributos Derivados

- Calculados a partir de outros
- Exemplo: idade (calculada da data de nascimento)

PESSOA			
string	id	PK	
string	nome		
date	data_nascimento		
int	idade		Derivado

Boas Práticas

1. Nomenclatura

- Use nomes significativos
- Mantenha padrão consistente
- Evite abreviações ambíguas

2. Tipos de Dados

- Escolha tipos apropriados
- Defina tamanhos adequados
- Considere restrições

3. Normalização

- Evite redundância
- Mantenha atomicidade

- Considere dependências

Exemplos Práticos

Sistema de Vendas

PRODUTO		
string	codigo	PK
string	nome	
string	descricao	
float	preco	
int	estoque	
string	categoria	FK
date	data_cadastro	
boolean	ativo	

Sistema de RH

FUNCIONARIO			
string	matricula	PK	
string	nome		
date	data_admissao		
float	salario_base		
float	bonus		Derivado
string	departamento	FK	
string[]	habilidades		
string	end_rua		
string	end_numero		
string	end_cidade		

Considerações Importantes

1. Integridade

- Defina restrições adequadas
- Valide valores permitidos

- Mantenha consistência

2. Performance

- Otimize tipos de dados
- Planeje índices
- Considere volume

3. Manutenibilidade

- Documente decisões
- Facilite evolução
- Mantenha simplicidade

Padrões Comuns

1. Atributos de Auditoria

ENTIDADE		
string	id	PK
string	nome	
date	criado_em	
string	criado_por	
date	alterado_em	
string	alterado_por	

2. Atributos de Status

PEDIDO		
string	numero	PK
string	status	
boolean	ativo	
date	data_status	

Conclusão

Atributos são fundamentais para:

- Descrição de dados
- Integridade do modelo
- Funcionalidade do sistema
- Qualidade da informação

A modelagem adequada de atributos:

- Facilita manutenção
- Melhora performance
- Garante consistência
- Suporta evolução

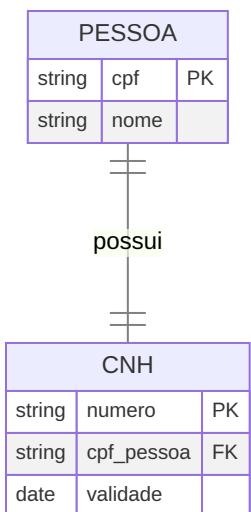
Cardinalidade

A cardinalidade é um conceito fundamental na modelagem ER que define o número de instâncias de uma entidade que podem se relacionar com instâncias de outra entidade.

Tipos de Cardinalidade

1. Um-para-Um (1:1)

- Cada instância de uma entidade está associada a no máximo uma instância da outra entidade
- Representada como: ||--||



Exemplos Práticos (1:1)

- Pessoa ↔ CPF
- Funcionário ↔ Matrícula
- País ↔ Capital

2. Um-para-Muitos (1:N)

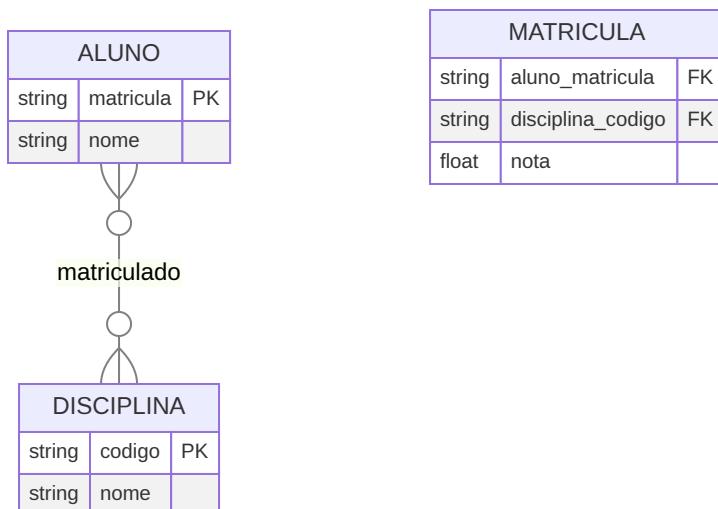
- Uma instância de uma entidade pode estar associada a várias instâncias da outra entidade
- Representada como: ||--o{ ou ||--|{

Exemplos Práticos (1:N)

- Departamento → Funcionários
- Cliente → Pedidos
- Professor → Turmas

3. Muitos-para-Muitos (N:M)

- Várias instâncias de uma entidade podem estar associadas a várias instâncias da outra entidade
- Representada como: }o--o{



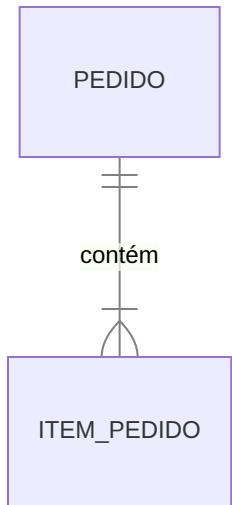
Exemplos Práticos (N:M)

- Alunos ↔ Disciplinas
- Produtos ↔ Fornecedores
- Autores ↔ Livros

Participação

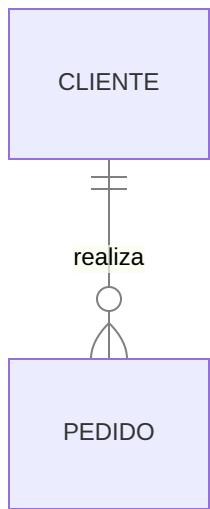
Participação Total

- Todas as instâncias da entidade participam do relacionamento
- Representada por linha dupla: ||



Participação Parcial

- Algumas instâncias podem não participar do relacionamento
- Representada por linha com círculo: o{



Restrições de Cardinalidade

Mínima

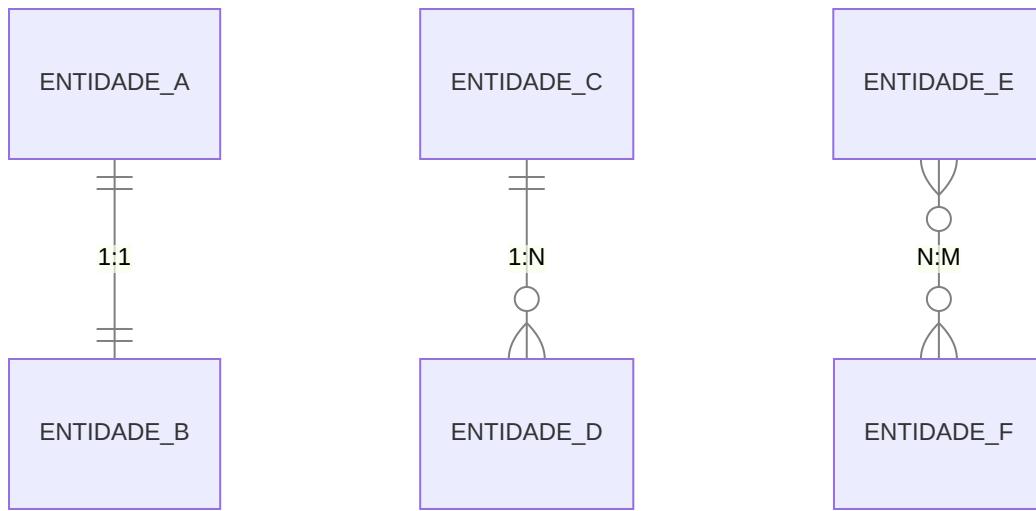
- Número mínimo de instâncias relacionadas
- Exemplo: zero (opcional) ou um (obrigatório)

Máxima

- Número máximo de instâncias relacionadas
- Exemplo: um ou muitos (n)

Notações Comuns

1. Notação Crow's Foot

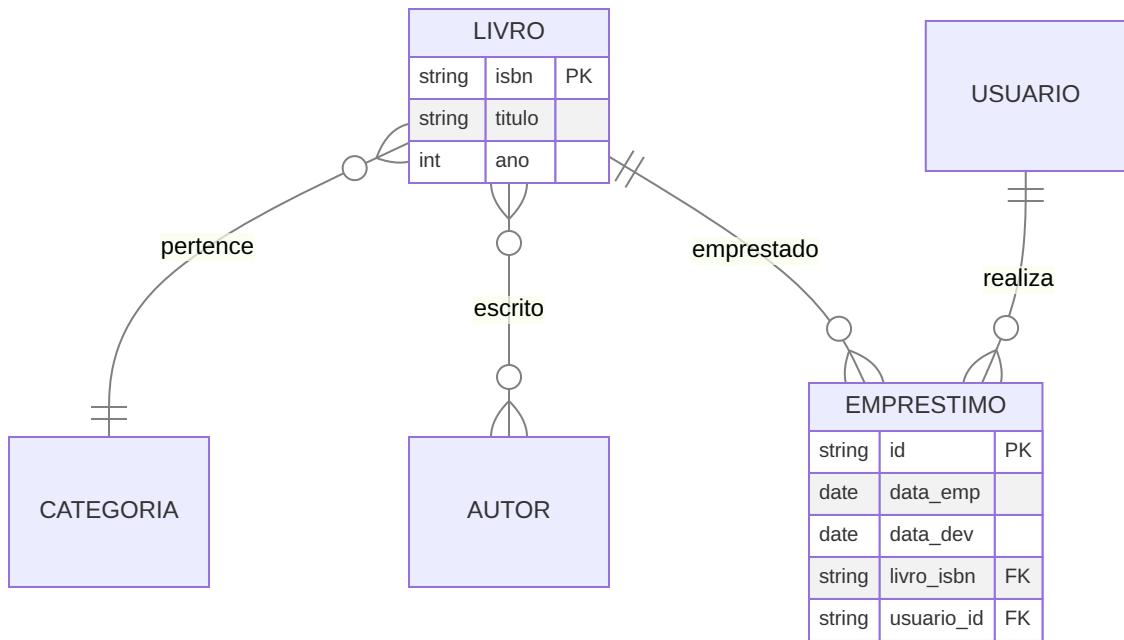


2. Notação Chen

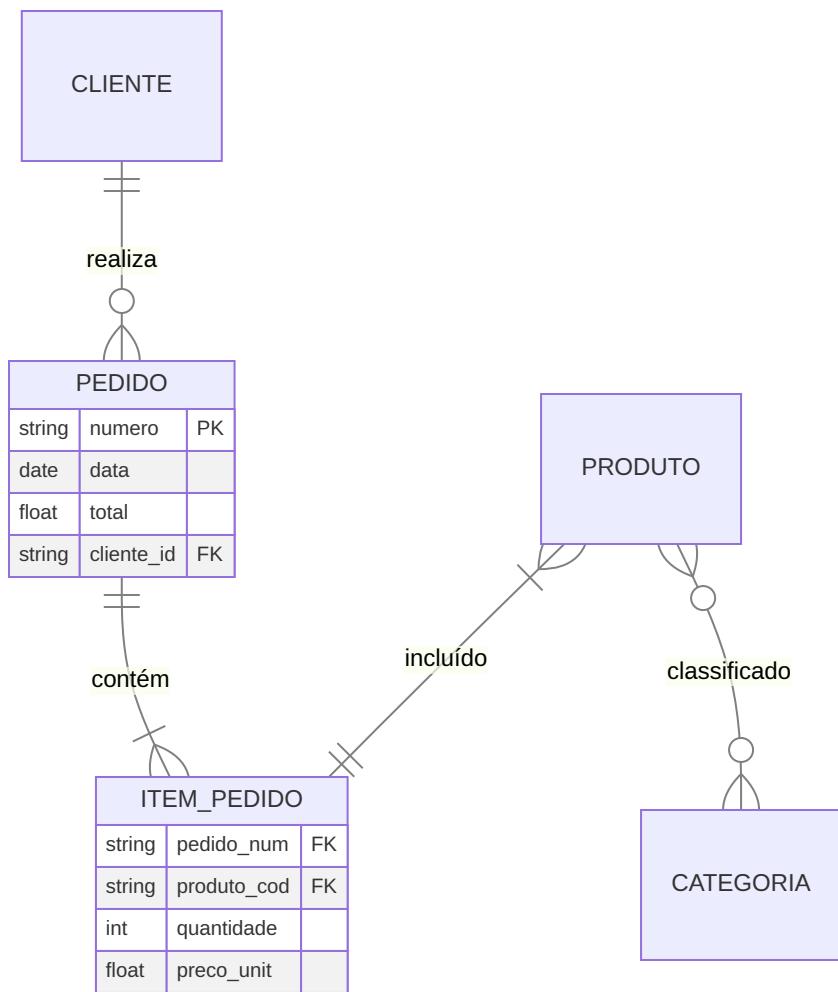
- (1,1) : Exatamente um
- (0,1) : Zero ou um
- (1,N) : Um ou mais
- (0,N) : Zero ou mais

Exemplos Complexos

Sistema de Biblioteca



Sistema de E-commerce



Boas Práticas

1. Análise de Requisitos

- Entenda regras de negócio
- Identifique restrições
- Valide com stakeholders

2. Modelagem

- Escolha cardinalidade apropriada
- Considere participação

- Documente decisões

3. Implementação

- Planeje chaves estrangeiras
- Defina índices adequados
- Implemente restrições

Considerações Importantes

1. Performance

- Impacto em consultas
- Estratégias de indexação
- Otimização de joins

2. Manutenibilidade

- Flexibilidade para mudanças
- Documentação clara
- Padrões consistentes

3. Integridade

- Regras de negócio
- Consistência dos dados
- Validações necessárias

Conclusão

A cardinalidade é essencial para:

- Estruturação correta dos dados
- Integridade do modelo
- Performance do sistema
- Manutenibilidade do código

Uma modelagem adequada de cardinalidade:

- Reflete regras de negócio
- Facilita implementação
- Previne problemas futuros
- Melhora qualidade dos dados

Restrições no Modelo ER

As restrições no modelo ER são regras que garantem a integridade e consistência dos dados. Elas definem limites e condições que os dados devem satisfazer.

Tipos de Restrições

1. Restrições de Chave

ENTIDADE			
string	id	PK	Chave Primária
string	codigo	UK	Chave Única
string	ref	FK	Chave Estrangeira

Chave Primária (PK)

- Identifica unicamente cada instância
- Não pode ser nula
- Não pode ser duplicada

Chave Estrangeira (FK)

- Referencia chave primária de outra entidade
- Mantém integridade referencial
- Pode ser nula (dependendo da participação)

Chave Única (UK)

- Garante valores únicos
- Pode ser nula (diferente da PK)
- Permite múltiplas por entidade

2. Restrições de Participação

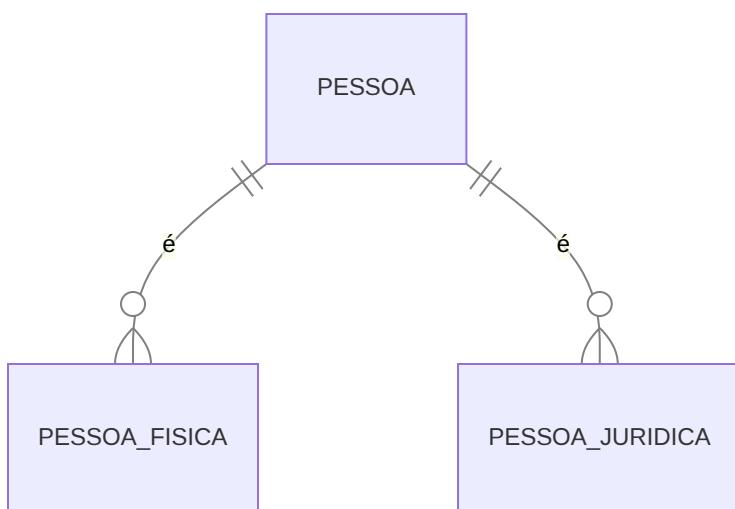
Participação Total

- Todas as instâncias devem participar
- Representada por linha dupla
- Exemplo: Todo item deve ter um produto

Participação Parcial

- Participação opcional
- Representada por linha simples
- Exemplo: Cliente pode não ter pedidos

3. Restrições de Sobreposição



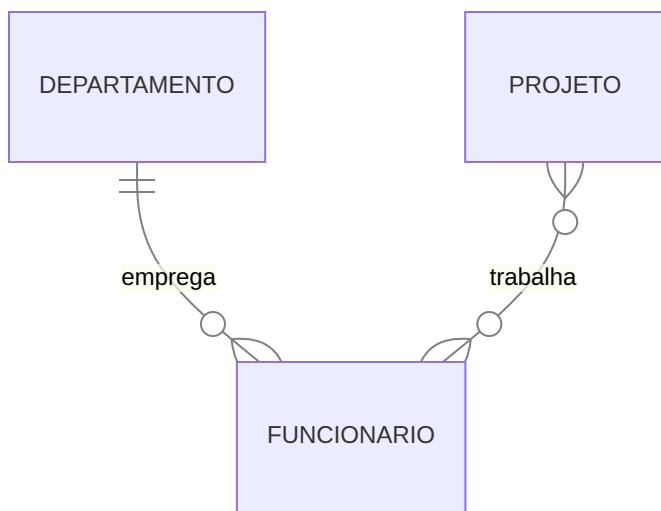
Disjunção (XOR)

- Entidade participa em apenas um relacionamento
- Mutuamente exclusivo
- Exemplo: Pessoa Física XOR Jurídica

Sobreposição

- Entidade pode participar em múltiplos relacionamentos
- Não exclusivo
- Exemplo: Funcionário pode ser Cliente

4. Restrições de Cardinalidade



Cardinalidade Mínima

- Número mínimo de participações
- Exemplo: Zero (opcional) ou Um (obrigatório)

Cardinalidade Máxima

- Número máximo de participações
- Exemplo: Um ou Muitos (N)

Implementação de Restrições

1. Nível de Banco de Dados

```
CREATE TABLE Produto (
    codigo VARCHAR(10) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    preco DECIMAL(10,2) CHECK (preco > 0)
);
```

2. Nível de Aplicação

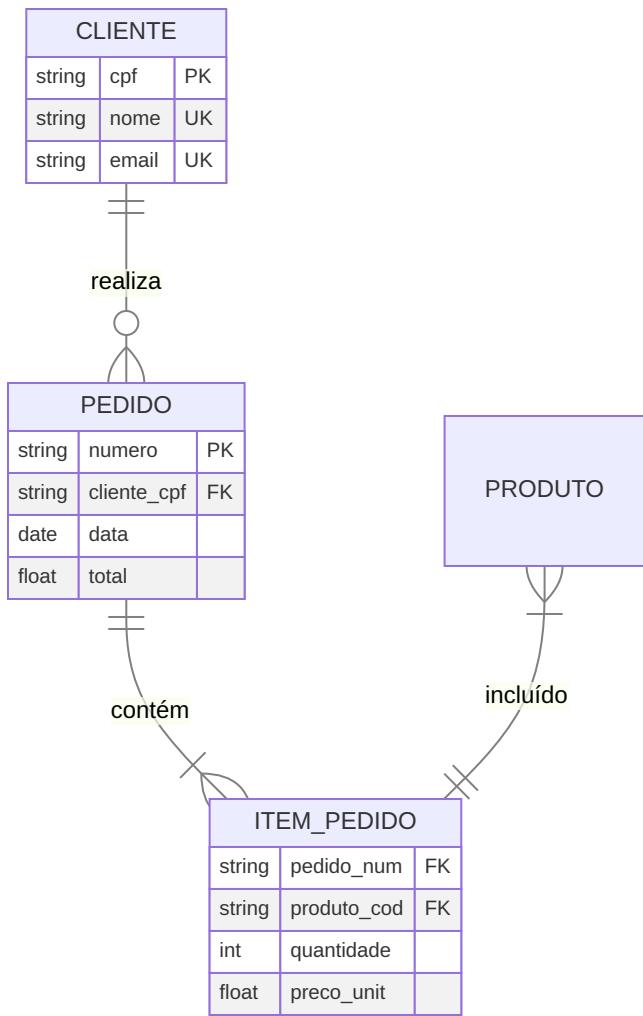
```
public class Produto {
    @Id
    private String codigo;

    @NotNull
    private String nome;

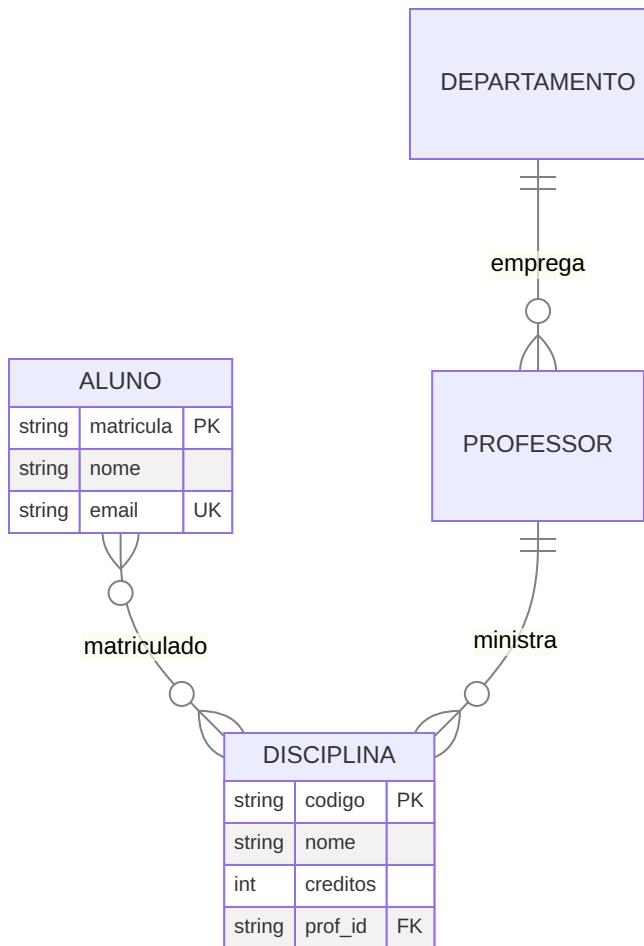
    @Positive
    private BigDecimal preco;
}
```

Exemplos Práticos

Sistema de Vendas



Sistema Acadêmico



Boas Práticas

1. Definição de Restrições

- Identifique regras de negócio
- Documente claramente
- Valide com stakeholders

2. Implementação

- Use mecanismos apropriados
- Mantenha consistência

- Considere performance

3. Manutenção

- Monitore violações
- Atualize quando necessário
- Mantenha documentação

Considerações Importantes

1. Performance

- Impacto das restrições
- Otimização de verificações
- Estratégias de indexação

2. Integridade

- Consistência dos dados
- Validações necessárias
- Tratamento de exceções

3. Flexibilidade

- Evolução do modelo
- Mudanças de requisitos
- Manutenibilidade

Padrões Comuns

1. Restrições de Negócio

CONTA			
string	numero	PK	
float	saldo		CHECK ≥ 0
string	status		CHECK IN ('ativo', 'inativo')

2. Restrições Temporais

CONTRATO			
string	id	PK	
date	inicio		CHECK $< \text{fim}$
date	fim		CHECK $> \text{inicio}$

Conclusão

Restrições são essenciais para:

- Integridade dos dados
- Regras de negócio
- Consistência do modelo
- Qualidade da informação

Uma modelagem adequada de restrições:

- Previne inconsistências
- Facilita manutenção
- Melhora confiabilidade
- Garante qualidade

Restrições de Chave

As restrições de chave são fundamentais para garantir a integridade e unicidade dos dados em um modelo ER.

Tipos de Chaves

1. Chave Primária (PK)

PRODUTO			
string	codigo	PK	Identificador único
string	nome		
float	preco		

Características

- Identifica unicamente cada registro
- Não pode conter valores nulos
- Deve ser imutável
- Pode ser simples ou composta

Exemplos

- CPF em uma tabela de clientes
- Número de matrícula de alunos
- ISBN para livros

2. Chave Estrangeira (FK)

Características

- Referencia uma chave primária
- Mantém integridade referencial

- Pode ser nula (relacionamento opcional)
- Pode participar de chave primária composta

Regras de Integridade

- Valor deve existir na tabela referenciada
- Atualização em cascata (opcional)
- Deleção em cascata (opcional)

3. Chave Única (UK)

USUARIO			
string	id	PK	
string	email	UK	Único por usuário
string	username	UK	Único no sistema

Características

- Garante unicidade do valor
- Pode conter nulos (diferente da PK)
- Múltiplas por entidade
- Útil para campos alternativos de busca

Implementação

1. SQL DDL

```
CREATE TABLE Cliente (
    cpf VARCHAR(11) PRIMARY KEY,
    email VARCHAR(100) UNIQUE,
    telefone VARCHAR(20) UNIQUE,
    CONSTRAINT valid_cpf CHECK (LENGTH(cpf) = 11)
);
```

```
CREATE TABLE Pedido (
    numero SERIAL PRIMARY KEY,
    cliente_cpf VARCHAR(11) REFERENCES Cliente(cpf),
    data DATE NOT NULL
);
```

2. Mapeamento Objeto-Relacional

```
@Entity
public class Cliente {
    @Id
    private String cpf;

    @Column(unique = true)
    private String email;

    @Column(unique = true)
    private String telefone;
}
```

Boas Práticas

1. Escolha de Chaves Primárias

- Prefira valores naturais e imutáveis
- Considere o tamanho do campo
- Evite chaves compostas complexas
- Use surrogate keys quando apropriado

2. Gestão de Chaves Estrangeiras

- Defina política de atualização/deleção
- Considere impacto na performance

- Planeje índices adequadamente
- Documente relacionamentos

3. Unicidade

- Identifique campos que exigem unicidade
- Considere unicidade combinada
- Planeje validações em múltiplas camadas
- Trate conflitos adequadamente

Padrões Comuns

1. Chave Natural vs Surrogate

PRODUTO_NATURAL			
string	codigo	PK	Chave natural
string	nome		

PRODUTO_SURROGATE			
int	id	PK	Chave surrogate
string	codigo	UK	Chave natural
string	nome		

2. Chave Composta

MATRICULA			
string	aluno_id	PK,FK	Parte 1 da PK
string	disciplina_id	PK,FK	Parte 2 da PK
string	semestre	PK	Parte 3 da PK
float	nota		

Considerações de Performance

1. Indexação

- Índices automáticos em PKs
- Índices opcionais em FKs

- Índices únicos para UKs
- Impacto em inserções/atualizações

2. Joins

- Otimização de consultas
- Cardinalidade das relações
- Estratégias de indexação
- Planos de execução

Conclusão

Restrições de chave são essenciais para:

- Garantir integridade dos dados
- Estabelecer relacionamentos
- Otimizar consultas
- Manter consistência

Uma implementação adequada:

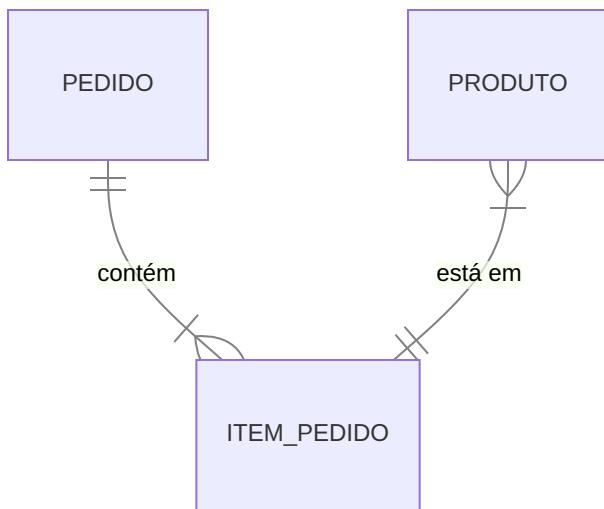
- Previne duplicidades
- Facilita manutenção
- Melhora performance
- Garante qualidade dos dados

Restrições de Participação

As restrições de participação definem como as entidades devem participar em relacionamentos, especificando regras de obrigatoriedade e cardinalidade.

Tipos de Participação

1. Participação Total (Obrigatória)



Características

- Toda instância da entidade deve participar do relacionamento
- Representada por linha dupla (||)
- Cardinalidade mínima maior que zero

Exemplos

- Todo item de pedido deve estar associado a um produto
- Todo funcionário deve pertencer a um departamento
- Toda conta bancária deve ter um titular

2. Participação Parcial (Opcional)

Características

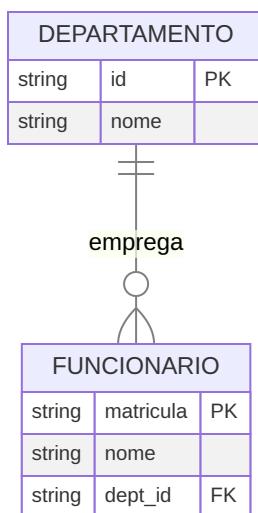
- Instâncias podem ou não participar do relacionamento
- Representada por linha simples
- Cardinalidade mínima igual a zero

Exemplos

- Um cliente pode não ter pedidos
- Um produto pode não ter categoria
- Um funcionário pode não ter projetos

Notações de Cardinalidade

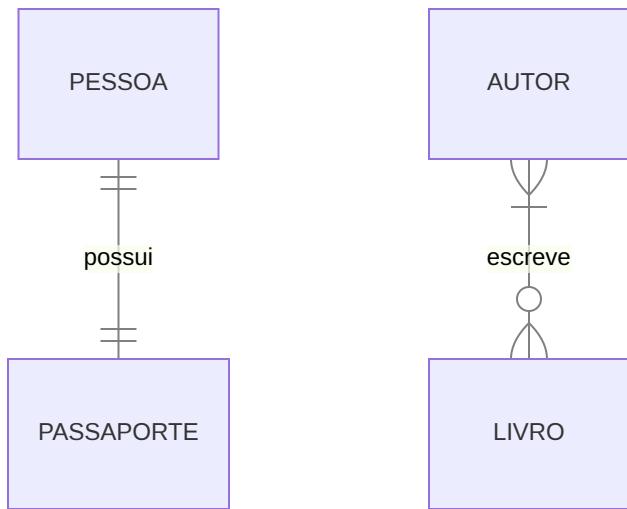
1. Cardinalidade Mínima



Tipos

- Zero (o): Participação opcional
- Um (|): Participação obrigatória

2. Cardinalidade Máxima



Tipos

- Um (): Máximo uma instância
- Muitos {}: Múltiplas instâncias

Implementação

1. Nível de Banco de Dados

```

-- Participação Total
CREATE TABLE Funcionario (
    matricula VARCHAR(10) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    departamento_id INTEGER NOT NULL,
    FOREIGN KEY (departamento_id)
    REFERENCES Departamento(id)
);

-- Participação Parcial
CREATE TABLE Produto (
    codigo VARCHAR(10) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    categoria_id INTEGER,
    FOREIGN KEY (categoria_id)
)
  
```

```
REFERENCES Categoria(id)
```

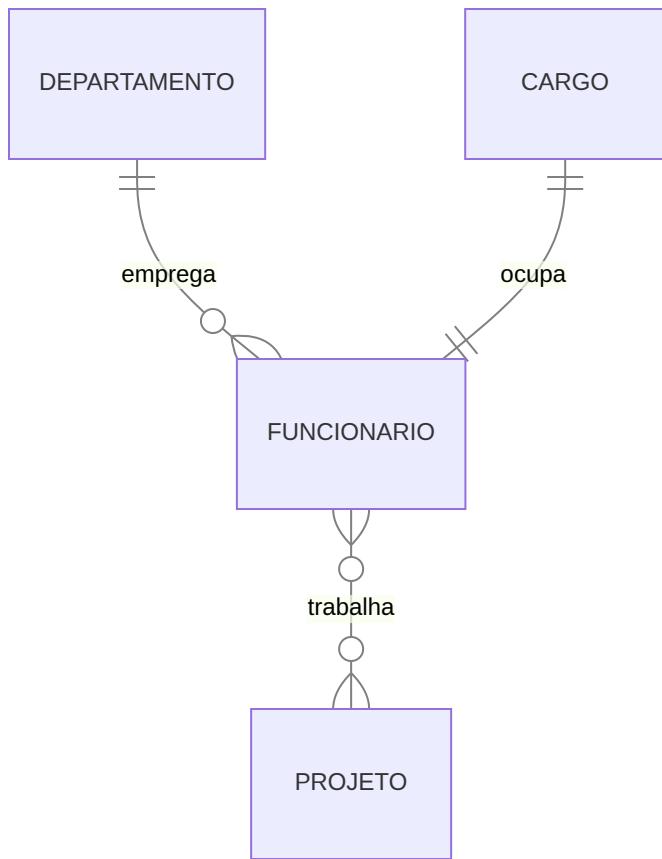
```
);
```

2. Nível de Aplicação

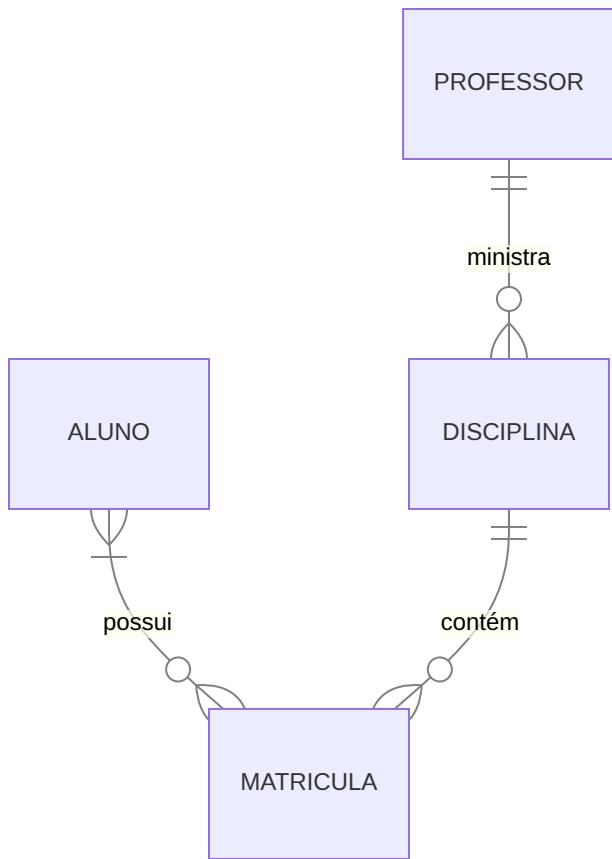
```
@Entity  
public class Funcionario {  
    @Id  
    private String matricula;  
  
    @NotNull  
    private String nome;  
  
    @ManyToOne(optional = false)  
    private Departamento departamento;  
}  
  
@Entity  
public class Produto {  
    @Id  
    private String codigo;  
  
    @NotNull  
    private String nome;  
  
    @ManyToOne(optional = true)  
    private Categoria categoria;  
}
```

Casos de Uso

1. Sistema de Recursos Humanos



2. Sistema Acadêmico



Boas Práticas

1. Modelagem

- Identifique claramente as regras de negócio
- Documente as restrições
- Valide com stakeholders
- Considere casos especiais

2. Implementação

- Use constraints apropriadas
- Implemente validações em múltiplas camadas
- Considere impacto na performance

- Mantenha consistência

3. Manutenção

- Monitore violações
- Atualize conforme necessário
- Mantenha documentação
- Revise periodicamente

Considerações Importantes

1. Impacto no Negócio

- Regras operacionais
- Processos de negócio
- Requisitos legais
- Flexibilidade vs. Controle

2. Performance

- Impacto em consultas
- Estratégias de indexação
- Otimização de joins
- Cache e performance

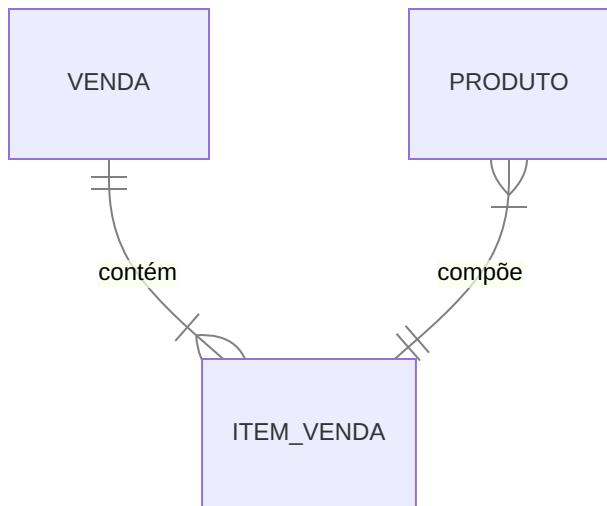
3. Manutenibilidade

- Evolução do sistema
- Mudanças de requisitos
- Documentação

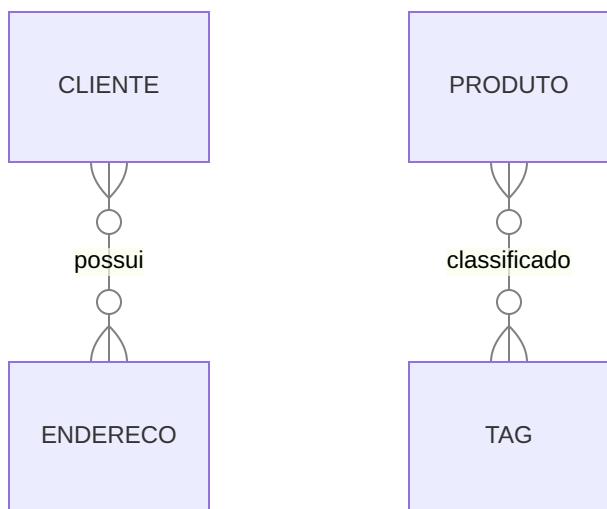
- Testes

Padrões Comuns

1. Relacionamentos Mandatórios



2. Relacionamentos Opcionais



Conclusão

Restrições de participação são cruciais para:

- Integridade dos dados

- Regras de negócio
- Consistência do modelo
- Qualidade da informação

Uma modelagem adequada:

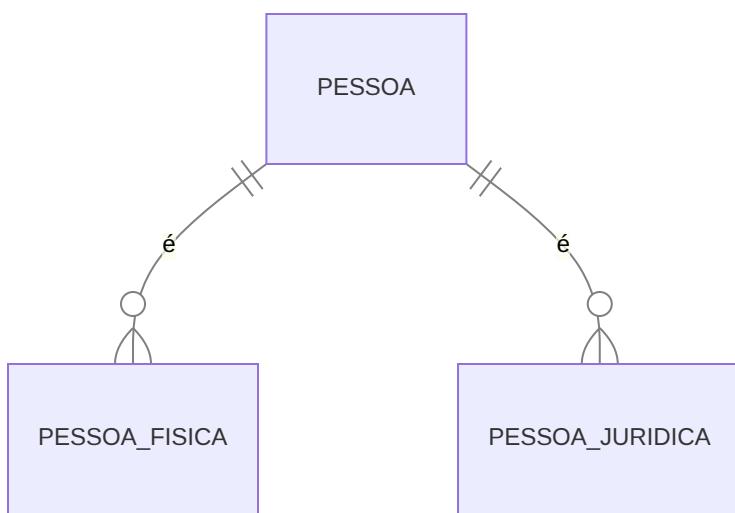
- Reflete requisitos reais
- Facilita implementação
- Melhora manutenibilidade
- Garante consistência

Restrições de Sobreposição

As restrições de sobreposição definem como entidades podem participar em múltiplos relacionamentos ou subtipos, especificando se essa participação pode ser simultânea ou deve ser exclusiva.

Tipos de Restrições

1. Disjunção (XOR)



Características

- Entidade participa em apenas um relacionamento/subtipo
- Mutuamente exclusivo
- Soma das participações = 1

Exemplos

- Uma pessoa só pode ser física OU jurídica
- Um funcionário só pode ser CLT OU PJ
- Uma conta só pode ser corrente OU poupança

2. Sobreposição (Overlap)

Características

- Entidade pode participar em múltiplos relacionamentos/subtipos
- Não exclusivo
- Participação simultânea permitida

Exemplos

- Uma pessoa pode ser cliente E funcionário
- Um professor pode lecionar em múltiplos departamentos
- Um produto pode pertencer a várias categorias

Implementação

1. Disjunção em SQL

```
CREATE TABLE Pessoa (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL
);

CREATE TABLE PessoaFisica (
    pessoa_id INTEGER PRIMARY KEY,
    cpf VARCHAR(11) UNIQUE NOT NULL,
    FOREIGN KEY (pessoa_id) REFERENCES Pessoa(id),
    CONSTRAINT unique_pessoa CHECK (
        NOT EXISTS (
            SELECT 1 FROM PessoaJuridica
            WHERE pessoa_id = PessoaFisica.pessoa_id
        )
    )
);
```

```

CREATE TABLE PessoaJuridica (
    pessoa_id INTEGER PRIMARY KEY,
    cnpj VARCHAR(14) UNIQUE NOT NULL,
    FOREIGN KEY (pessoa_id) REFERENCES Pessoa(id),
    CONSTRAINT unique_pessoa CHECK (
        NOT EXISTS (
            SELECT 1 FROM PessoaFisica
            WHERE pessoa_id = PessoaJuridica.pessoa_id
        )
    )
);

```

2. Sobreposição em SQL

```

CREATE TABLE Pessoa (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL
);

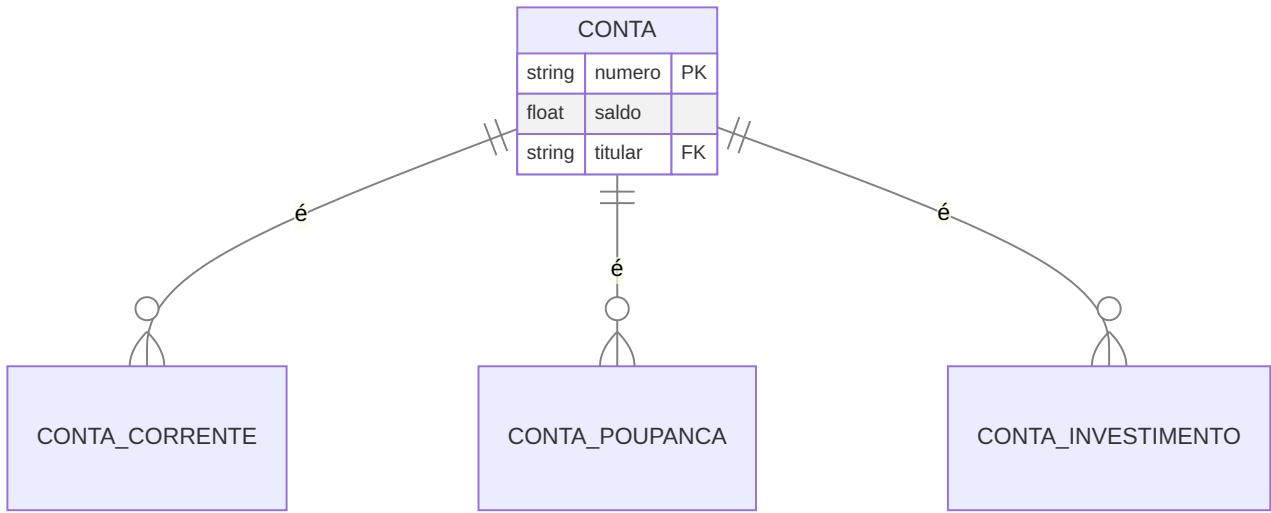
CREATE TABLE Cliente (
    pessoa_id INTEGER PRIMARY KEY,
    codigo_cliente VARCHAR(10) UNIQUE NOT NULL,
    FOREIGN KEY (pessoa_id) REFERENCES Pessoa(id)
);

CREATE TABLE Funcionario (
    pessoa_id INTEGER PRIMARY KEY,
    matricula VARCHAR(10) UNIQUE NOT NULL,
    FOREIGN KEY (pessoa_id) REFERENCES Pessoa(id)
);

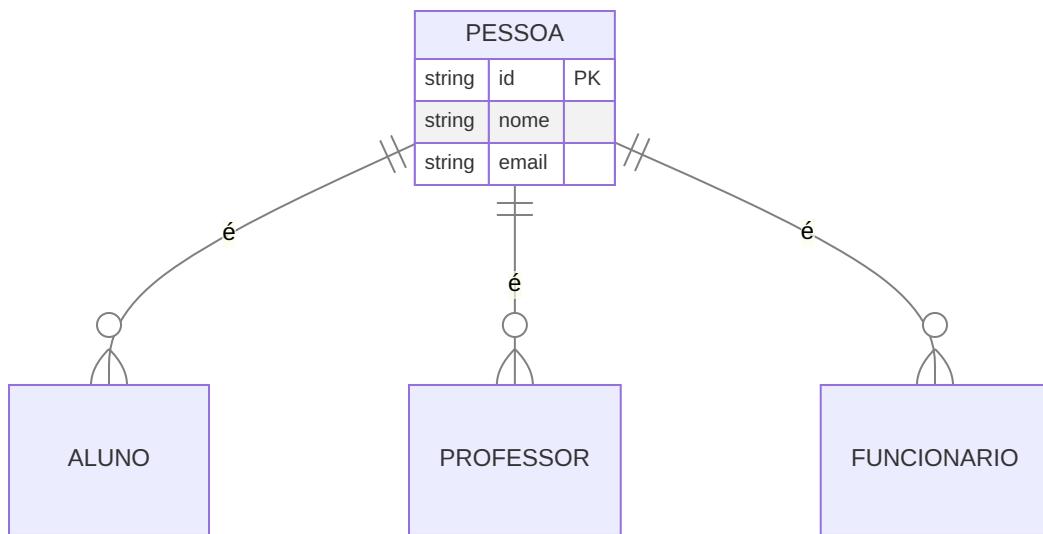
```

Casos de Uso

1. Sistema Bancário



2. Sistema Acadêmico



Boas Práticas

1. Modelagem

- Identifique claramente as regras de exclusividade
- Documente as restrições
- Valide com stakeholders
- Considere evolução futura

2. Implementação

- Use constraints apropriadas
- Implemente validações em múltiplas camadas
- Considere performance
- Mantenha consistência

3. Manutenção

- Monitore violações
- Atualize conforme necessidade
- Mantenha documentação
- Revise periodicamente

Considerações Importantes

1. Performance

- Impacto das verificações de constraints
- Estratégias de indexação
- Otimização de consultas
- Cache e performance

2. Flexibilidade

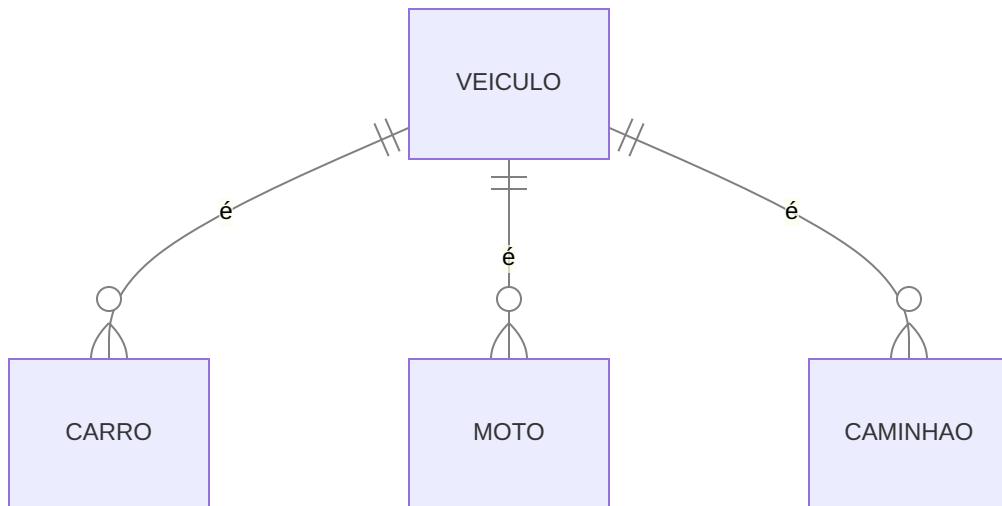
- Mudanças nas regras de negócio
- Evolução do sistema
- Migração de dados
- Manutenibilidade

3. Integridade

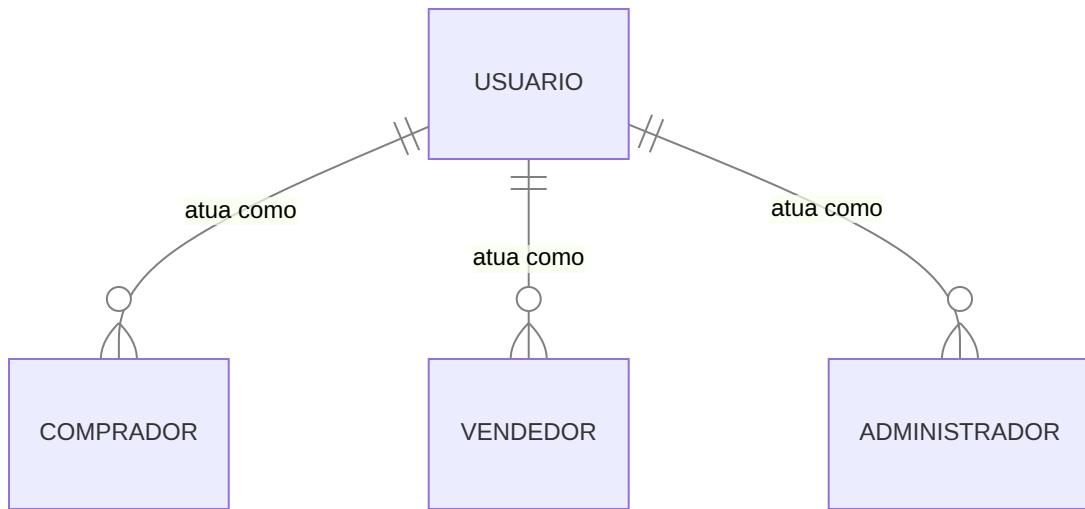
- Consistência dos dados
- Validações
- Tratamento de erros
- Recuperação de falhas

Padrões Comuns

1. Herança Exclusiva



2. Papéis Múltiplos



Conclusão

Restrições de sobreposição são essenciais para:

- Integridade do modelo
- Regras de negócio
- Consistência dos dados
- Qualidade da informação

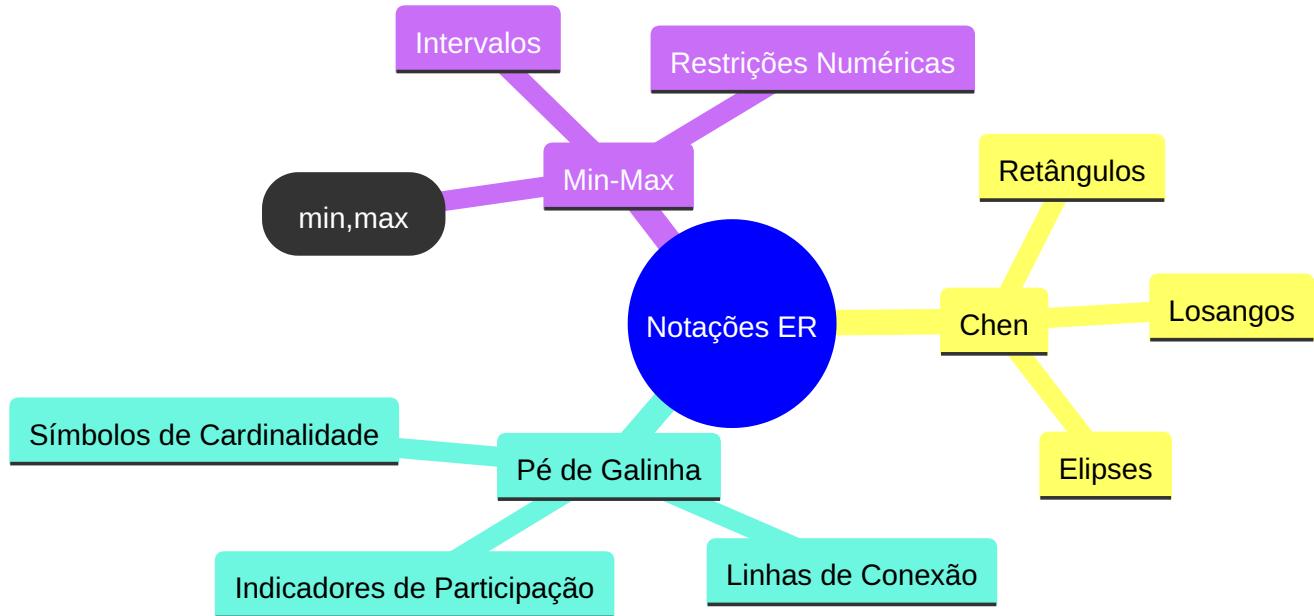
Uma modelagem adequada:

- Reflete requisitos reais
- Facilita implementação
- Melhora manutenibilidade
- Garante consistência

Notações do Modelo ER

O Modelo Entidade-Relacionamento (ER) pode ser representado usando diferentes notações. Cada notação tem suas particularidades e é adequada para diferentes contextos.

Visão Geral das Notações



Comparação das Notações

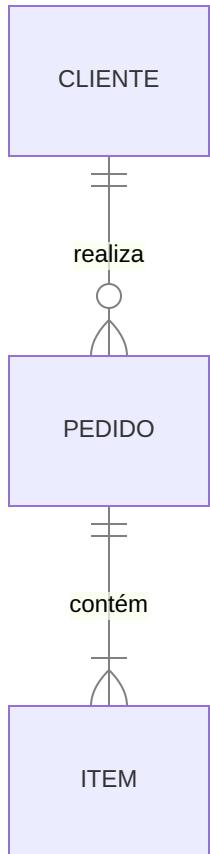
1. Elementos Básicos

Elemento	Chen	Pé de Galinha	Min-Max
Entidade	Retângulo	Retângulo	Retângulo
Relacionamento	Losango	Linha	Linha
Atributo	Elipse	Texto	Texto
Cardinalidade	Texto (1,N,M)	Símbolos	(min,max)

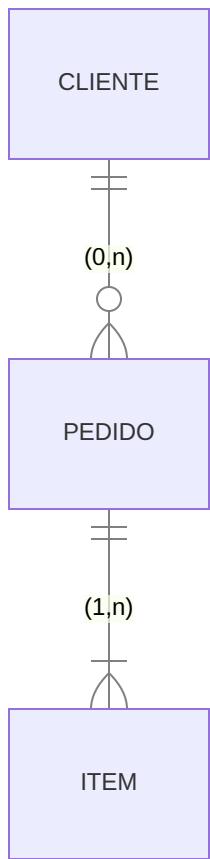
2. Representação de Cardinalidade

Notação de Chen

Notação Pé de Galinha



Notação Min-Max



Características das Notações

1. Notação de Chen

Vantagens

- Clara e intuitiva
- Boa para aprendizado
- Detalhada

Desvantagens

- Ocupa muito espaço
- Pode ficar confusa em modelos grandes
- Menos usada em ferramentas modernas

2. Notação Pé de Galinha

Vantagens

- Compacta
- Amplamente usada
- Suportada por muitas ferramentas

Desvantagens

- Símbolos podem ser confusos inicialmente
- Menos detalhada que Chen
- Variações entre ferramentas

3. Notação Min-Max

Vantagens

- Precisa
- Flexível
- Boa para restrições complexas

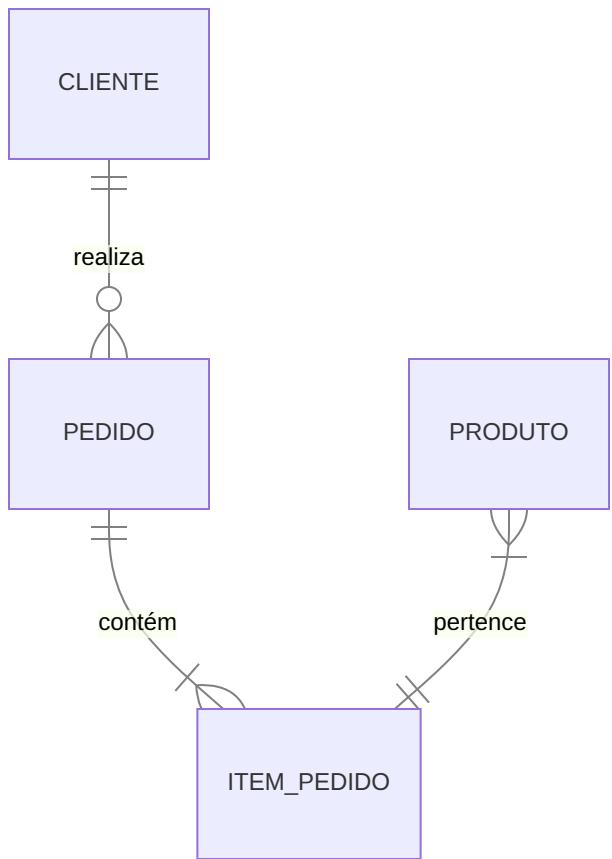
Desvantagens

- Mais complexa
- Menos intuitiva
- Requer mais explicação

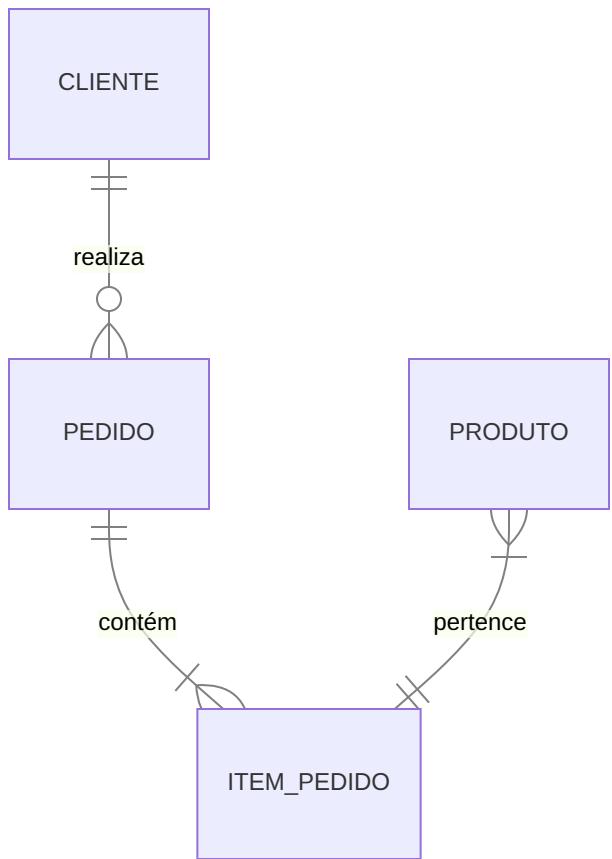
Exemplos Práticos

1. Sistema de Vendas

Chen

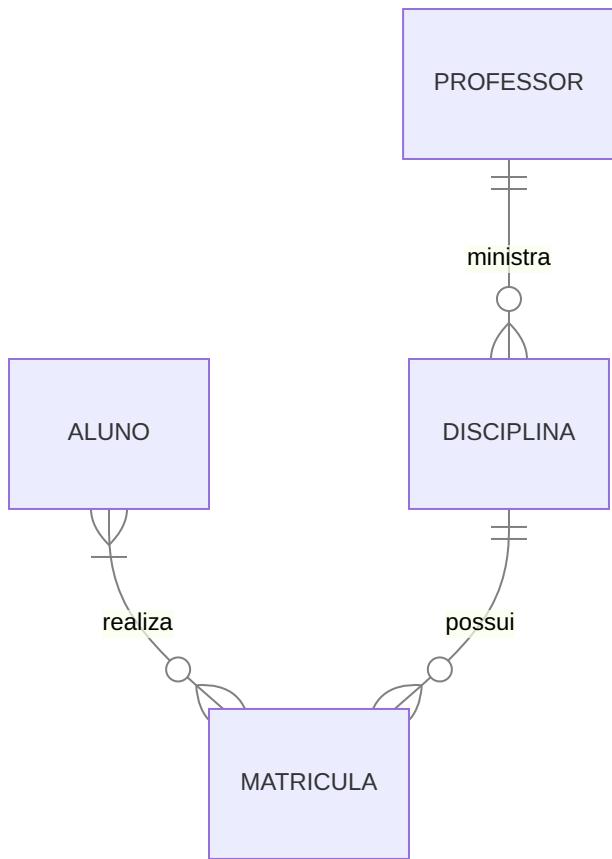


Pé de Galinha

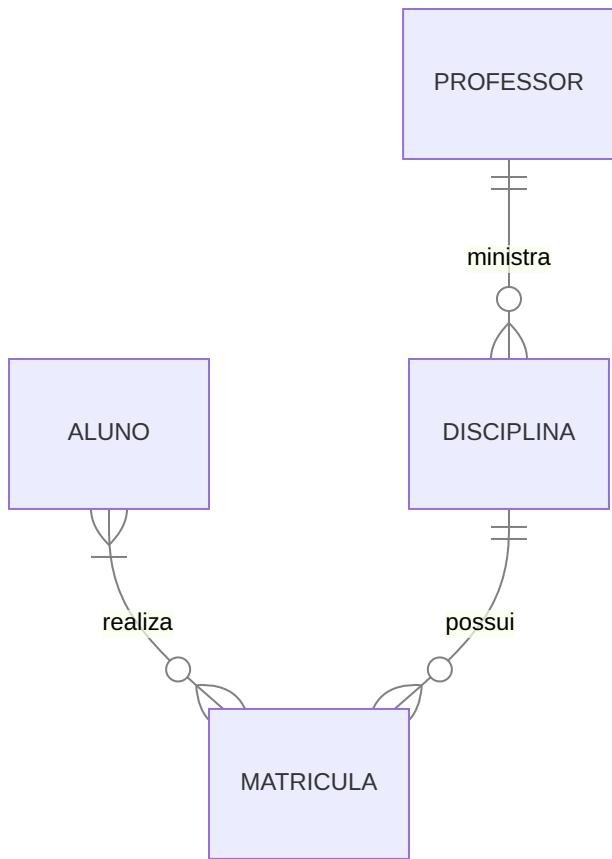


2. Sistema Acadêmico

Chen



Pé de Galinha



Escolhendo a Notação Adequada

1. Fatores a Considerar

- Público-alvo
- Complexidade do modelo
- Ferramentas disponíveis
- Padrões organizacionais
- Necessidade de detalhamento

2. Recomendações

Para Ensino

- Use notação de Chen

- Explique todos os elementos
- Mostre exemplos simples

Para Projetos

- Use notação Pé de Galinha
- Mantenha consistência
- Documente convenções

Para Especificações

- Use notação Min-Max
- Detalhe restrições
- Forneça exemplos

Ferramentas e Suporte

1. Ferramentas Populares

- Draw.io
- Lucidchart
- Visual Paradigm
- MySQL Workbench
- Enterprise Architect

2. Recursos Online

- Editores online
- Plugins para IDEs
- Geradores de documentação

- Conversores entre notações

Conclusão

A escolha da notação ER deve considerar:

- Necessidades do projeto
- Público-alvo
- Ferramentas disponíveis
- Padrões organizacionais

Independente da notação:

- Mantenha consistência
- Documente convenções
- Priorize clareza
- Considere manutenibilidade

Notação de Chen

A notação de Chen, desenvolvida por Peter Chen em 1976, é uma das notações mais tradicionais e didáticas para modelagem Entidade-Relacionamento (ER).

Elementos Básicos

1. Entidades

CLIENTE		
string	id	PK
string	nome	
string	email	

- Representadas por retângulos
- Nome em MAIÚSCULAS
- Singular
- Substantivos

2. Relacionamentos

- Representados por losangos
- Verbos no presente
- MAIÚSCULAS
- Conectam entidades

3. Atributos

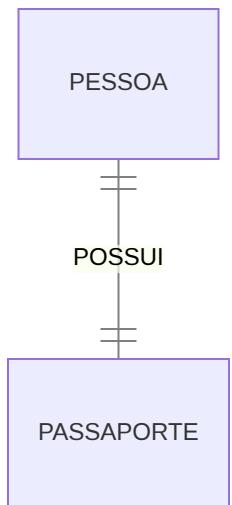
PRODUTO			
string	codigo	PK	Chave Primária
string	nome		Nome do produto
float	preco		Preço unitário
string	descricao		Descrição detalhada

Tipos de Atributos

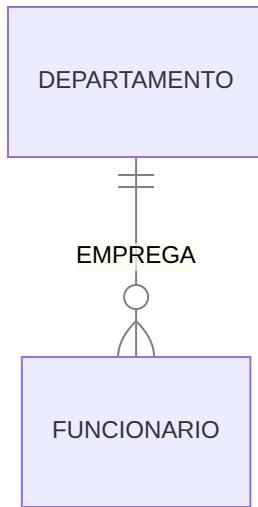
- **Simples:** valor único
- **Compostos:** múltiplos componentes
- **Multivalorados:** múltiplos valores
- **Derivados:** calculados
- **Chave:** identificador único

Cardinalidade

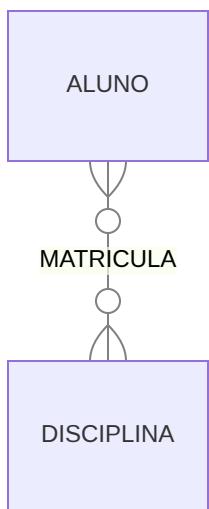
1. Um-para-Um (1:1)



2. Um-para-Muitos (1:N)

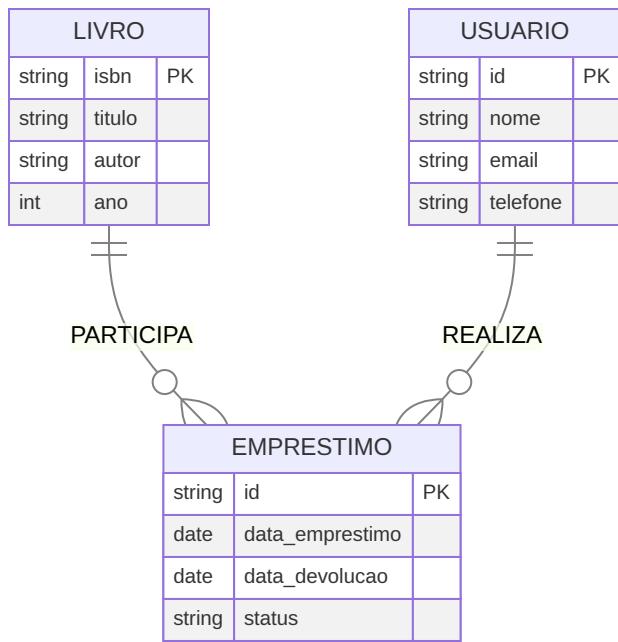


3. Muitos-para-Muitos (N:M)

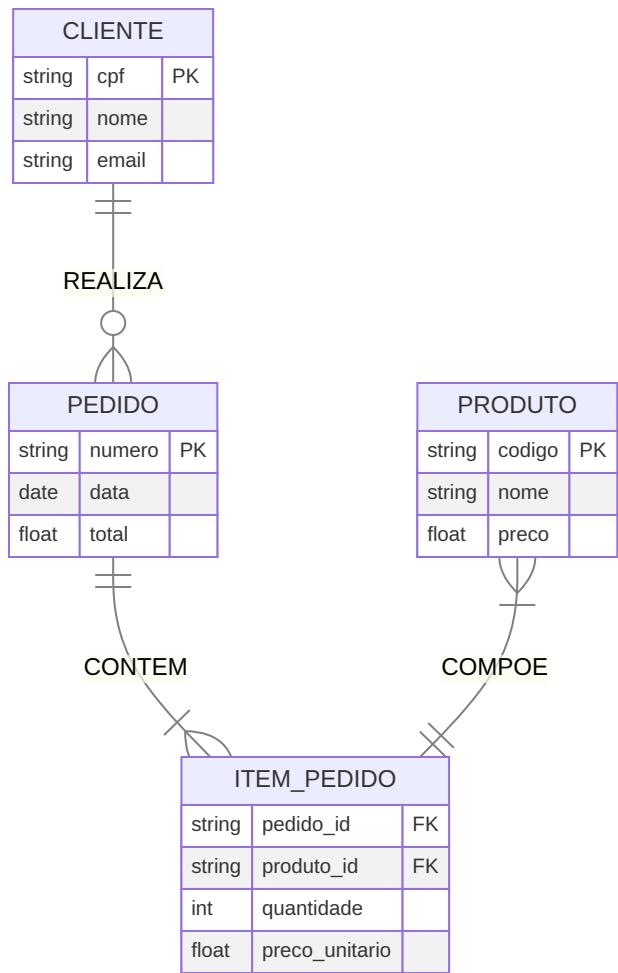


Exemplos Detalhados

1. Sistema de Biblioteca



2. Sistema de Vendas



Regras de Modelagem

1. Entidades

- Nomes significativos
- Singular
- MAIÚSCULAS
- Substantivos

2. Relacionamentos

- Verbos no presente

- MAIÚSCULAS
- Direção clara
- Significado preciso

3. Atributos

- Nomes descritivos
- Tipos apropriados
- Domínios definidos
- Chaves identificadas

Boas Práticas

1. Nomenclatura

- Consistente
- Clara
- Significativa
- Padronizada

2. Layout

- Organizado
- Espaçado
- Legível
- Hierárquico

3. Documentação

- Completa

- Atualizada
- Acessível
- Versionada

Vantagens da Notação Chen

1. Clareza

- Elementos distintos
- Fácil leitura
- Visual intuitivo
- Sem ambiguidade

2. Didática

- Ótima para ensino
- Fácil aprendizado
- Conceitos claros
- Exemplos simples

3. Detalhamento

- Atributos visíveis
- Cardinalidade explícita
- Relacionamentos claros
- Restrições evidentes

Limitações

1. Espaço

- Diagramas grandes
- Muitos elementos
- Layout complexo
- Difícil organização

2. Complexidade

- Modelos extensos
- Muitos atributos
- Relacionamentos complexos
- Manutenção trabalhosa

Ferramentas de Suporte

1. Software Específico

- Draw.io
- Lucidchart
- Visual Paradigm
- ERDPlus
- MySQL Workbench

2. Recursos Online

- Editores web
- Plugins IDE
- Geradores
- Conversores

Conclusão

A notação de Chen é ideal para:

- Ensino de modelagem
- Documentação detalhada
- Comunicação clara
- Projetos didáticos

Pontos-chave:

- Clareza visual
- Padrões consistentes
- Documentação completa
- Manutenção regular

Notação Pé de Galinha (Crow's Foot)

A notação Pé de Galinha, também conhecida como Crow's Foot, é uma das notações mais populares para modelagem de dados, especialmente em ferramentas CASE e ambientes profissionais.

Elementos Básicos

1. Entidades

CLIENTE		
string	id	PK
string	nome	
string	email	

- Representadas por retângulos
- Nome em MAIÚSCULAS
- Atributos listados internamente
- Chaves indicadas (PK/FK)

2. Relacionamentos

- Representados por linhas
- Cardinalidade nas extremidades
- Verbos conectando entidades
- Direção de leitura indicada

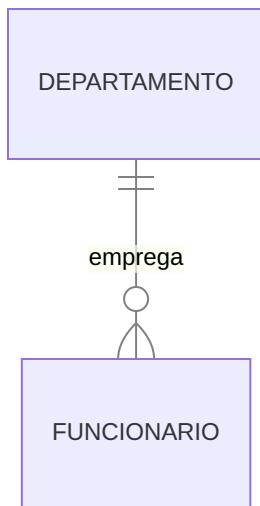
3. Símbolos de Cardinalidade

Símbolo	Significado
\ \	Exatamente um
\ o	Zero ou um
\ \{	Um ou mais
o{	Zero ou mais

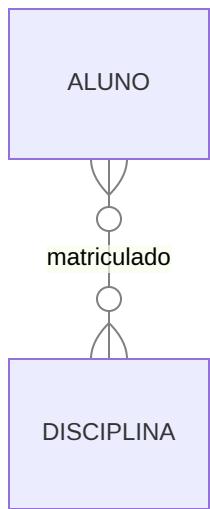
Exemplos de Cardinalidade

1. Um-para-Um (1:1)

2. Um-para-Muitos (1:N)

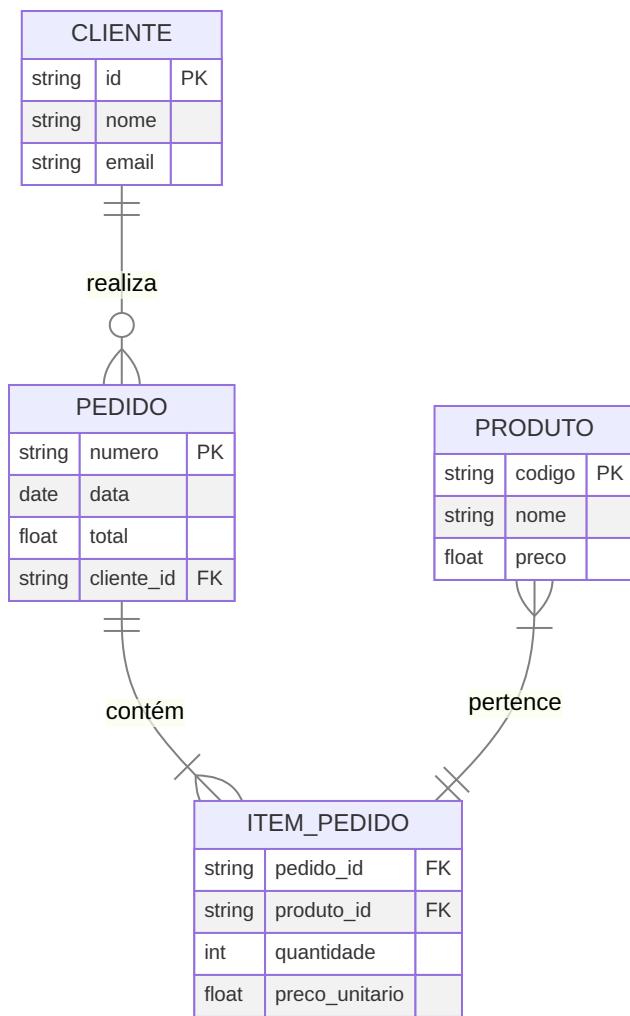


3. Muitos-para-Muitos (N:M)

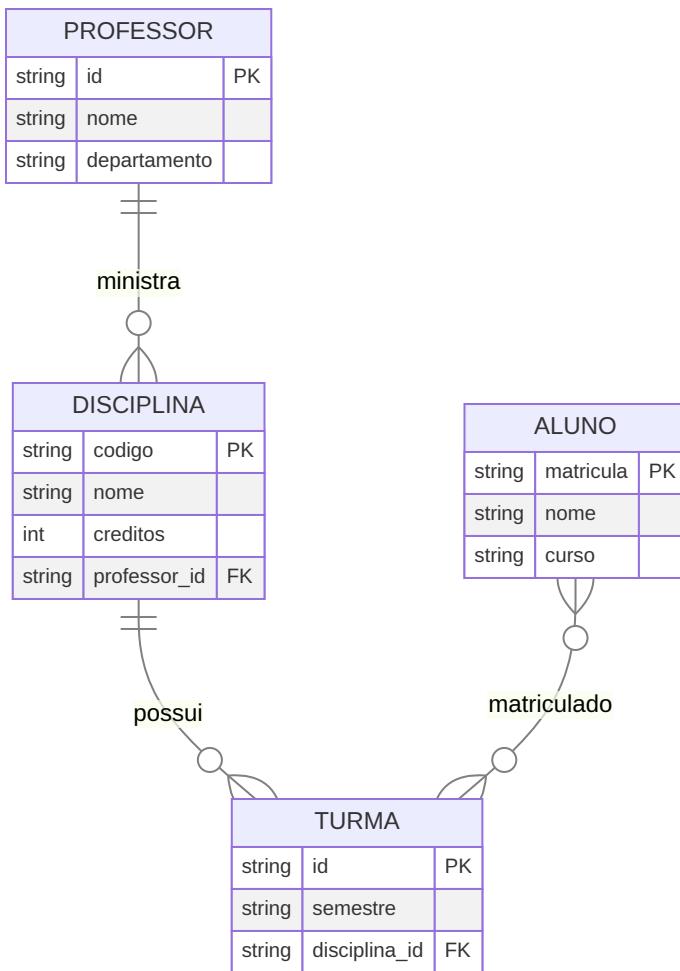


Modelos Complexos

1. Sistema de Vendas



2. Sistema Acadêmico



Vantagens

1. Compacta

- Economia de espaço
- Diagramas limpos
- Fácil visualização
- Escalável

2. Profissional

- Amplamente adotada
- Suporte em ferramentas

- Padrão de mercado
- Fácil integração

3. Prática

- Rápida de desenhar
- Fácil de modificar
- Clara de entender
- Simples de manter

Boas Práticas

1. Layout

- Minimize cruzamentos de linhas
- Alinhe entidades relacionadas
- Mantenha espaçamento consistente
- Organize por grupos lógicos

2. Nomenclatura

- Use nomes significativos
- Mantenha padrão consistente
- Evite abreviações
- Use termos do domínio

3. Relacionamentos

- Indique direção de leitura
- Use verbos significativos

- Evite redundância
- Mantenha simplicidade

Ferramentas Populares

1. Modelagem

- MySQL Workbench
- ERwin
- Visio
- Draw.io
- Lucidchart

2. Recursos

- Templates prontos
- Validação automática
- Geração de código
- Documentação integrada

Comparação com Outras Notações

1. Vantagens sobre Chen

- Mais compacta
- Mais moderna
- Mais utilizada
- Melhor suporte

2. Vantagens sobre UML

- Foco em dados
- Mais simples
- Mais específica
- Melhor para BD

Casos de Uso

1. Modelagem de Dados

- Bancos relacionais
- Data warehouses
- Sistemas OLTP
- Sistemas OLAP

2. Documentação

- Especificações
- Manuais técnicos
- Documentação API
- Modelos conceituais

Dicas Práticas

1. Início do Projeto

- Identifique entidades principais
- Estabeleça relacionamentos básicos

- Defina cardinalidades
- Valide com stakeholders

2. Manutenção

- Mantenha documentação atualizada
- Revise periodicamente
- Refatore quando necessário
- Versione alterações

Conclusão

A notação Pé de Galinha é:

- Eficiente
- Profissional
- Bem suportada
- Amplamente adotada

Recomendada para:

- Projetos profissionais
- Documentação técnica
- Modelagem de dados
- Comunicação entre equipes

Notação Min-Max

A notação Min-Max é uma abordagem precisa para representar cardinalidades em modelos de dados, usando pares ordenados (min,max) para especificar restrições de participação.

Conceitos Básicos

1. Formato

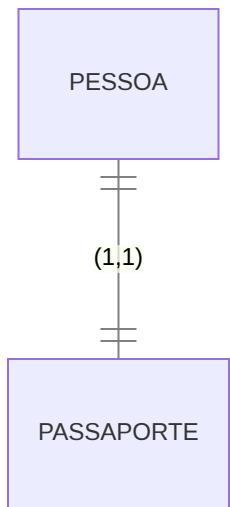
- (min,max) onde:
 - min: participação mínima
 - max: participação máxima
 - n: representa infinito

2. Valores Comuns

Notação	Significado
(0,1)	Zero ou um
(1,1)	Exatamente um
(0,n)	Zero ou mais
(1,n)	Um ou mais
(m,n)	Mínimo m, máximo n

Exemplos Básicos

1. Relacionamento Um-para-Um



Significado:

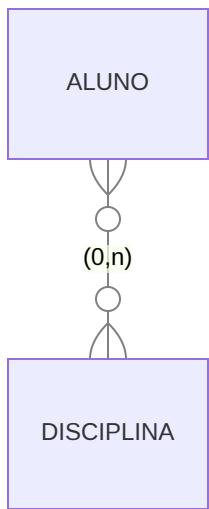
- Pessoa tem exatamente um passaporte
- Passaporte pertence a exatamente uma pessoa

2. Relacionamento Um-para-Muitos

Significado:

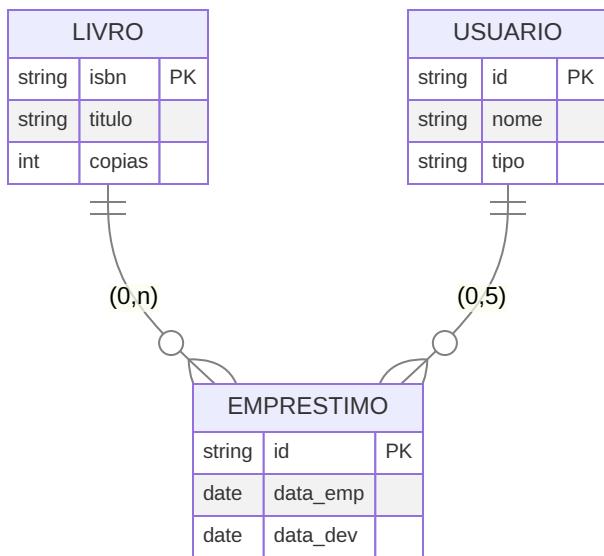
- Departamento pode ter zero ou mais funcionários
- Funcionário pertence a exatamente um departamento

3. Relacionamento Muitos-para-Muitos



Exemplos Complexos

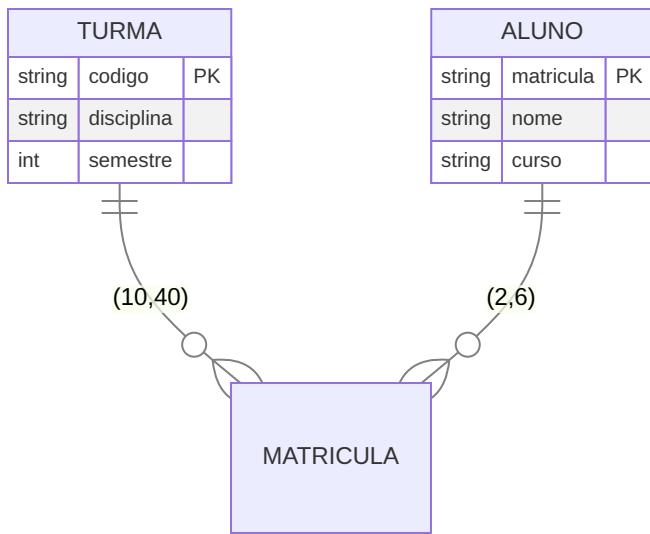
1. Sistema de Biblioteca



Restrições:

- Livro pode ter 0 ou n empréstimos
- Usuário pode ter 0 a 5 empréstimos simultâneos

2. Sistema de Matrícula



Restrições:

- Turma deve ter entre 10 e 40 alunos
- Aluno deve cursar entre 2 e 6 disciplinas

Vantagens

1. Precisão

- Restrições exatas
- Limites claros
- Regras explícitas
- Validação precisa

2. Flexibilidade

- Intervalos personalizados
- Restrições complexas
- Regras de negócio

- Validações específicas

3. Documentação

- Requisitos claros
- Especificações detalhadas
- Regras documentadas
- Manutenção facilitada

Casos de Uso

1. Regras de Negócio

- Limites de participação
- Restrições operacionais
- Políticas organizacionais
- Requisitos regulatórios

2. Validações

- Integridade de dados
- Consistência
- Conformidade
- Qualidade

Implementação

1. Banco de Dados

```
CREATE TABLE Turma (
    codigo VARCHAR(10) PRIMARY KEY,
```

```

disciplina VARCHAR(100),
semestre INTEGER,
CONSTRAINT chk_alunos
CHECK (num_alunos >= 10 AND num_alunos <= 40)
);

CREATE TABLE Matricula (
aluno_id VARCHAR(10),
turma_id VARCHAR(10),
CONSTRAINT chk_matriculas_por_aluno
CHECK (
(SELECT COUNT(*)
FROM Matricula
WHERE aluno_id = NEW.aluno_id) <= 6
)
);

```

2. Validação em Código

```

public class Turma {
    private List<Aluno> alunos;

    public void adicionarAluno(Aluno aluno) {
        if (alunos.size() >= 40) {
            throw new LimiteExcedidoException(
                "Turma não pode ter mais que 40 alunos");
        }
        alunos.add(aluno);
    }

    public boolean isValida() {
        return alunos.size() >= 10 && alunos.size() <= 40;
    }
}

```

Boas Práticas

1. Modelagem

- Defina limites realistas
- Documente justificativas
- Valide com stakeholders
- Mantenha consistência

2. Implementação

- Implemente validações
- Monitore limites
- Trate exceções
- Mantenha logs

3. Manutenção

- Revise periodicamente
- Ajuste conforme necessário
- Atualize documentação
- Monitore impactos

Ferramentas de Suporte

1. Modelagem

- ERwin
- MySQL Workbench
- Visual Paradigm
- Enterprise Architect

2. Validação

- Frameworks ORM
- Validadores de schema
- Ferramentas de teste
- Monitores de integridade

Conclusão

A notação Min-Max é ideal para:

- Especificações precisas
- Regras complexas
- Validações rigorosas
- Documentação detalhada

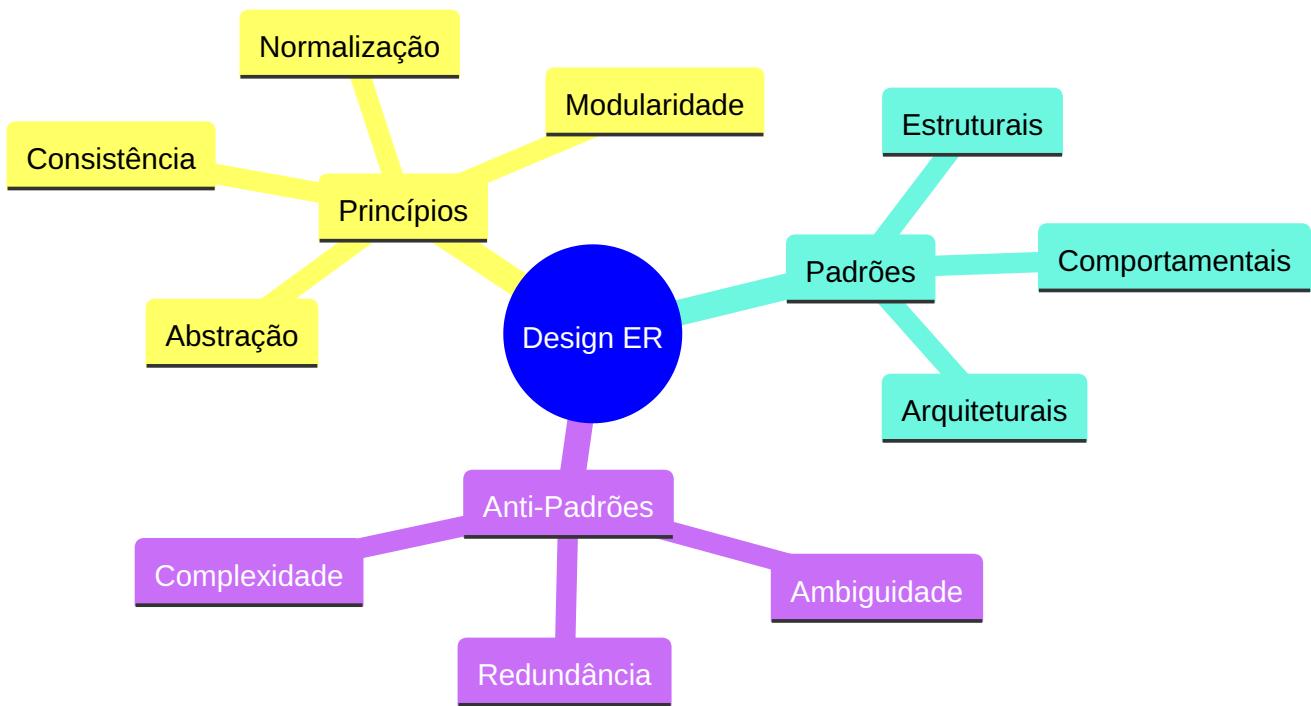
Benefícios principais:

- Clareza
- Precisão
- Flexibilidade
- Manutenibilidade

Design de Modelos Entidade-Relacionamento

O design de modelos Entidade-Relacionamento (ER) é uma habilidade crítica que combina princípios teóricos com experiência prática para criar modelos de dados eficientes e sustentáveis.

Visão Geral do Design ER



Processo de Design

1. Análise de Requisitos

- Identificação de entidades
- Definição de relacionamentos
- Levantamento de restrições
- Validação com stakeholders

2. Modelagem Conceitual

- Criação do modelo inicial
- Refinamento iterativo
- Validação de conceitos
- Documentação de decisões

3. Otimização

- Análise de desempenho
- Refinamento de estruturas
- Validação de padrões
- Eliminação de anti-padrões

Aspectos Críticos

1. Escalabilidade

- Crescimento de dados
- Evolução do schema
- Performance sustentável
- Manutenibilidade

2. Integridade

- Consistência dos dados
- Regras de negócio
- Restrições de domínio
- Validações

3. Usabilidade

- Clareza do modelo
- Facilidade de consulta
- Simplicidade de manutenção
- Documentação efetiva

Ferramentas e Técnicas

1. Modelagem

- Diagramas ER
- Ferramentas CASE
- Validadores de modelo
- Geradores de documentação

2. Validação

- Revisões por pares
- Testes de conceito
- Prototipação
- Benchmarking

Conclusão

O design ER efetivo requer:

- Conhecimento sólido dos princípios
- Compreensão dos padrões comuns

- Reconhecimento de anti-padrões
- Prática constante e iteração

Para aprofundamento, consulte:

- Design Principles ([Princípios de Design ER](#))
- Common Patterns ([Padrões Comuns em Modelagem ER](#))
- Anti-Patterns ([Anti-Padrões em Modelagem ER](#))

Princípios de Design ER

Os princípios de design ER são diretrizes fundamentais que orientam a criação de modelos de dados eficientes, manutáveis e escaláveis.

Princípios Fundamentais

1. Abstração Adequada

- Nível correto de detalhamento
 - Identificação das informações essenciais
 - Eliminação de detalhes supérfluos
 - Equilíbrio entre simplicidade e completude
 - Foco nas necessidades do negócio
- Representação clara do domínio
 - Alinhamento com conceitos do negócio
 - Vocabulário consistente com stakeholders
 - Mapeamento direto de processos
 - Facilidade de compreensão
- Balanceamento de complexidade
 - Decomposição de estruturas complexas
 - Agrupamento lógico de elementos
 - Gerenciamento de dependências
 - Simplicidade sem perda de funcionalidade
- Foco nos aspectos relevantes
 - Priorização de requisitos críticos

- Identificação de casos de uso principais
- Suporte a objetivos do negócio
- Flexibilidade para evolução

2. Normalização Apropriada

- **Eliminação de redundância**
 - Identificação de dados duplicados
 - Consolidação de informações
 - Estruturas normalizadas
 - Exceções justificadas
- **Integridade dos dados**
 - Consistência das informações
 - Regras de validação
 - Restrições de integridade
 - Garantias de qualidade
- **Eficiência de armazenamento**
 - Otimização de estruturas
 - Uso adequado de tipos de dados
 - Estratégias de compressão
 - Gerenciamento de espaço
- **Facilidade de manutenção**
 - Simplicidade de atualizações
 - Minimização de impactos
 - Clareza nas modificações

- Rastreabilidade de mudanças

3. Modularidade

- **Decomposição lógica**

- Separação de conceitos
- Agrupamento funcional
- Interfaces bem definidas
- Limites claros

- **Coesão entre elementos**

- Relacionamentos significativos
- Dependências justificadas
- Agrupamentos naturais
- Minimização de fragmentação

- **Acoplamento controlado**

- Interfaces bem definidas
- Dependências minimizadas
- Isolamento de mudanças
- Flexibilidade de evolução

- **Reusabilidade**

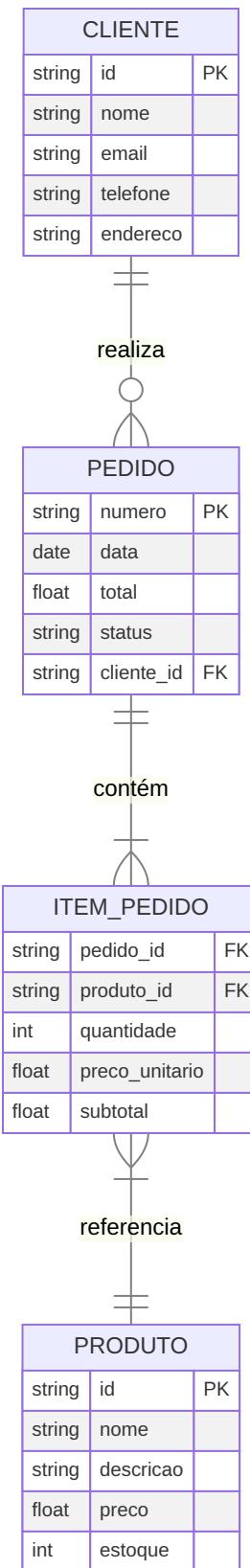
- Componentes genéricos
- Padrões reutilizáveis
- Estruturas flexíveis
- Adaptabilidade

4. Consistência

- **Nomenclatura padronizada**
 - Convenções de nomes
 - Termos do domínio
 - Abreviações consistentes
 - Clareza semântica
- **Convenções de modelagem**
 - Padrões de design
 - Práticas estabelecidas
 - Diretrizes documentadas
 - Conformidade com standards
- **Documentação uniforme**
 - Descrições claras
 - Metadados completos
 - Histórico de decisões
 - Justificativas de design
- **Regras consistentes**
 - Políticas uniformes
 - Restrições padronizadas
 - Validações coerentes
 - Tratamento de exceções

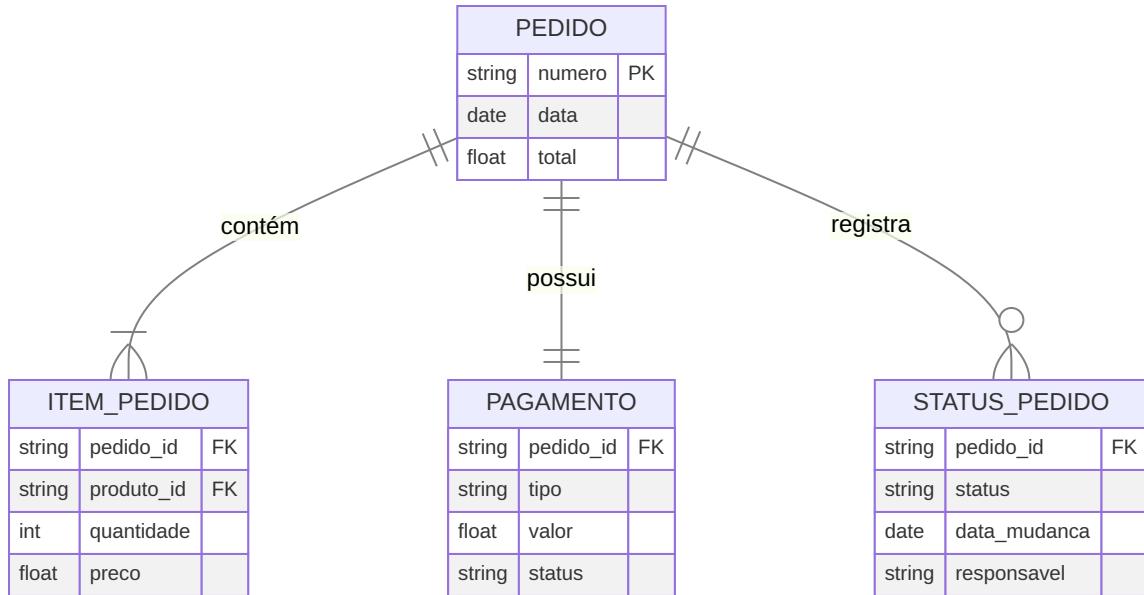
Exemplos Práticos

1. Abstração Adequada



2. Normalização

3. Modularidade



Aplicação dos Princípios

1. Na Modelagem Inicial

- Identificar entidades principais
 - Análise do domínio
 - Levantamento de requisitos
 - Definição de escopo
 - Validação com stakeholders
- Estabelecer relacionamentos básicos
 - Conexões naturais
 - Cardinalidades
 - Dependências

- Restrições
- **Definir atributos essenciais**
 - Dados fundamentais
 - Tipos apropriados
 - Restrições básicas
 - Documentação inicial
- **Validar normalização**
 - Análise de dependências
 - Eliminação de redundância
 - Verificação de integridade
 - Ajustes necessários

2. No Refinamento

- **Otimizar estruturas**
 - Análise de performance
 - Ajustes de design
 - Melhorias de eficiência
 - Validação de mudanças
- **Ajustar cardinalidades**
 - Revisão de relacionamentos
 - Verificação de restrições
 - Correção de anomalias
 - Documentação de mudanças
- **Refinar atributos**

- Revisão de tipos
 - Ajuste de restrições
 - Adição de metadados
 - Validação de regras
- **Validar consistência**
- Verificação de padrões
 - Teste de integridade
 - Análise de impacto
 - Documentação atualizada

Checklist de Validação

1. Abstração

- [] Nível adequado de detalhamento
 - [] Informações essenciais identificadas
 - [] Detalhes supérfluos eliminados
 - [] Complexidade gerenciável
 - [] Alinhamento com necessidades
- [] Representação clara do domínio
 - [] Conceitos bem definidos
 - [] Vocabulário consistente
 - [] Processos mapeados
 - [] Entendimento facilitado
- [] Complexidade gerenciável

- [] Estruturas decompostas
- [] Agrupamentos lógicos
- [] Dependências claras
- [] Simplicidade mantida
- [] Foco nos aspectos relevantes
 - [] Requisitos críticos atendidos
 - [] Casos de uso suportados
 - [] Objetivos alcançados
 - [] Flexibilidade preservada

2. Normalização

- [] Eliminação de redundância
 - [] Dados consolidados
 - [] Duplicações removidas
 - [] Estruturas otimizadas
 - [] Exceções documentadas
- [] Dependências funcionais corretas
 - [] Relacionamentos válidos
 - [] Integridade mantida
 - [] Anomalias eliminadas
 - [] Consistência garantida
- [] Integridade referencial
 - [] Chaves apropriadas
 - [] Relacionamentos válidos

- [] Restrições definidas
- [] Cascatas configuradas
- [] Eficiência de armazenamento
 - [] Tipos otimizados
 - [] Espaço gerenciado
 - [] Performance adequada
 - [] Recursos otimizados

Conclusão

Os princípios de design ER são fundamentais para:

- Qualidade do modelo de dados
- Eficiência do sistema
- Manutenibilidade do código
- Escalabilidade da solução

Sua aplicação consistente resulta em:

- Modelos mais robustos
- Sistemas mais confiáveis
- Manutenção simplificada
- Melhor documentação
- Maior satisfação dos usuários
- Menor custo total de propriedade

Padrões Comuns em Modelagem ER

Os padrões de modelagem ER são soluções comprovadas para problemas recorrentes no design de bancos de dados.

Categorias de Padrões

1. Padrões Estruturais

- Organização de entidades
- Relacionamentos comuns
- Hierarquias
- Composições

2. Padrões Comportamentais

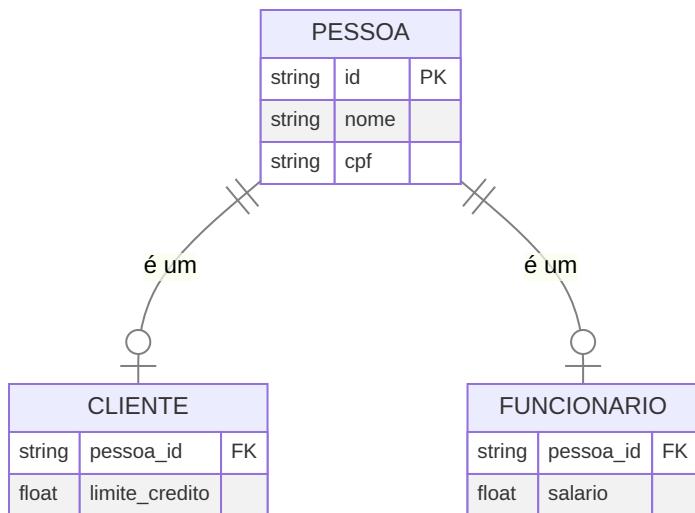
- Histórico de mudanças
- Workflows
- Estados e transições
- Auditoria

3. Padrões Arquiteturais

- Particionamento
- Distribuição
- Replicação
- Cache

Padrões Estruturais Comuns

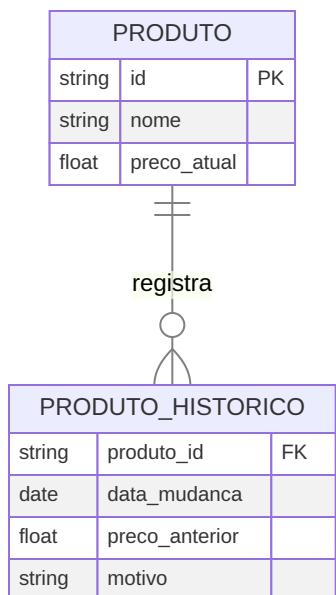
1. Hierarquia de Tipos



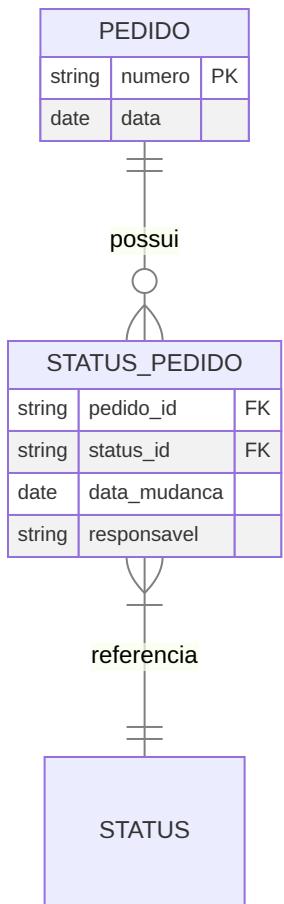
2. Composição

Padrões Comportamentais

1. Histórico de Mudanças

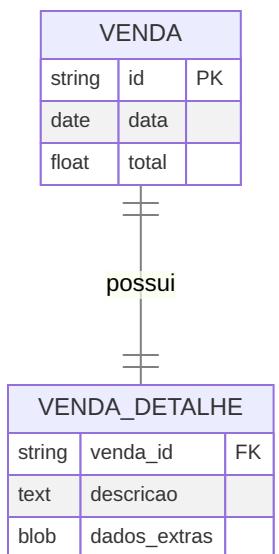


2. Workflow

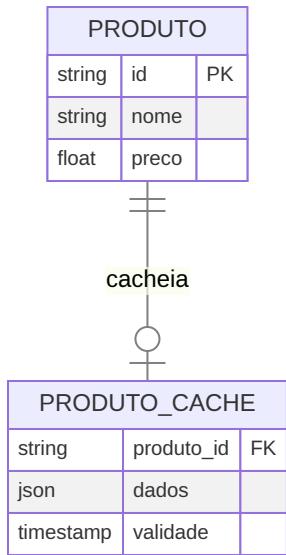


Padrões Arquiteturais

1. Particionamento



2. Cache



Implementação dos Padrões

1. Seleção

- Análise do problema
- Avaliação de alternativas
- Consideração de trade-offs
- Validação da escolha

2. Adaptação

- Customização para contexto
- Ajustes necessários
- Validação de mudanças
- Documentação

3. Integração

- Combinação de padrões
- Resolução de conflitos
- Otimização global
- Testes integrados

Benefícios

1. Desenvolvimento

- Soluções provadas
- Menor tempo de design
- Menos erros
- Melhor manutenção

2. Performance

- Otimizações conhecidas
- Escalabilidade planejada
- Eficiência comprovada
- Previsibilidade

3. Manutenção

- Padrões conhecidos
- Documentação padrão
- Facilidade de mudança
- Menor complexidade

Conclusão

Os padrões ER oferecem:

- Soluções testadas
- Economia de tempo
- Qualidade comprovada
- Base para inovação

Sua aplicação resulta em:

- Designs robustos
- Implementações eficientes
- Manutenção facilitada
- Sistemas escaláveis

Anti-Padrões em Modelagem ER

Anti-padrões são soluções comuns mas problemáticas que devem ser evitadas no design de modelos ER.

Tipos de Anti-Padrões

1. Estruturais

- Redundância excessiva
- Normalização inadequada
- Relacionamentos ambíguos
- Chaves mal definidas

2. Comportamentais

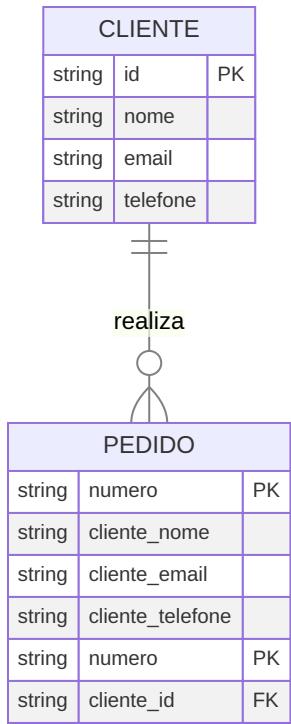
- Violação de integridade
- Inconsistência de dados
- Falta de atomicidade
- Problemas de concorrência

3. Arquiteturais

- Acoplamento excessivo
- Falta de escalabilidade
- Complexidade desnecessária
- Falta de flexibilidade

Anti-Padrões Comuns

1. Redundância de Dados



2. Relacionamentos Ambíguos

Problemas e Consequências

1. Redundância

- Inconsistência de dados
- Anomalias de atualização
- Desperdício de espaço
- Complexidade de manutenção

2. Ambiguidade

- Dificuldade de entendimento
- Erros de implementação

- Problemas de integridade
- Complexidade de queries

3. Desnormalização Excessiva

- Duplicação de dados
- Inconsistências
- Dificuldade de manutenção
- Performance comprometida

Como Identificar

1. Sinais de Alerta

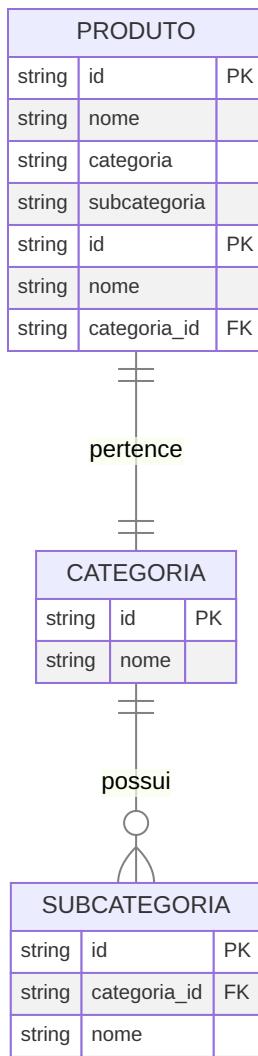
- Dados duplicados
- Relacionamentos circulares
- Atributos multivalorados
- Dependências parciais

2. Análise de Impacto

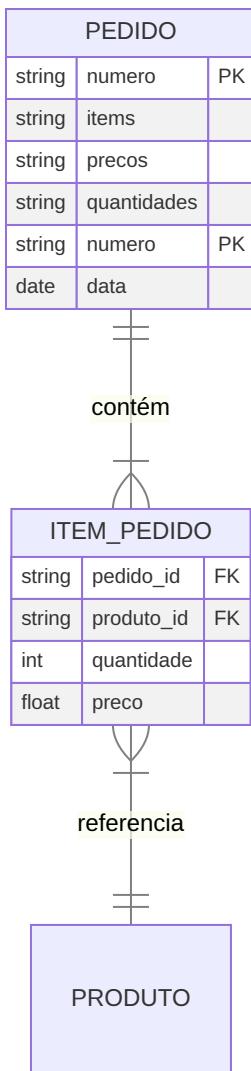
- Performance degradada
- Dificuldade de mudança
- Erros frequentes
- Complexidade crescente

Como Corrigir

1. Refatoração



2. Normalização



Prevenção

1. Boas Práticas

- Modelagem cuidadosa
- Revisão por pares
- Testes adequados
- Documentação clara

2. Validação Contínua

- Análise de performance
- Revisão de estrutura
- Monitoramento de uso
- Feedback de usuários

Checklist de Revisão

1. Estrutura

- [] Ausência de redundância
- [] Normalização adequada
- [] Relacionamentos claros
- [] Chaves bem definidas

2. Comportamento

- [] Integridade garantida
- [] Consistência mantida
- [] Atomicidade respeitada
- [] Concorrência tratada

3. Arquitetura

- [] Acoplamento controlado
- [] Escalabilidade possível
- [] Complexidade justificada
- [] Flexibilidade mantida

Conclusão

Evitar anti-padrões é crucial para:

- Qualidade do sistema
- Manutenibilidade
- Performance
- Escalabilidade

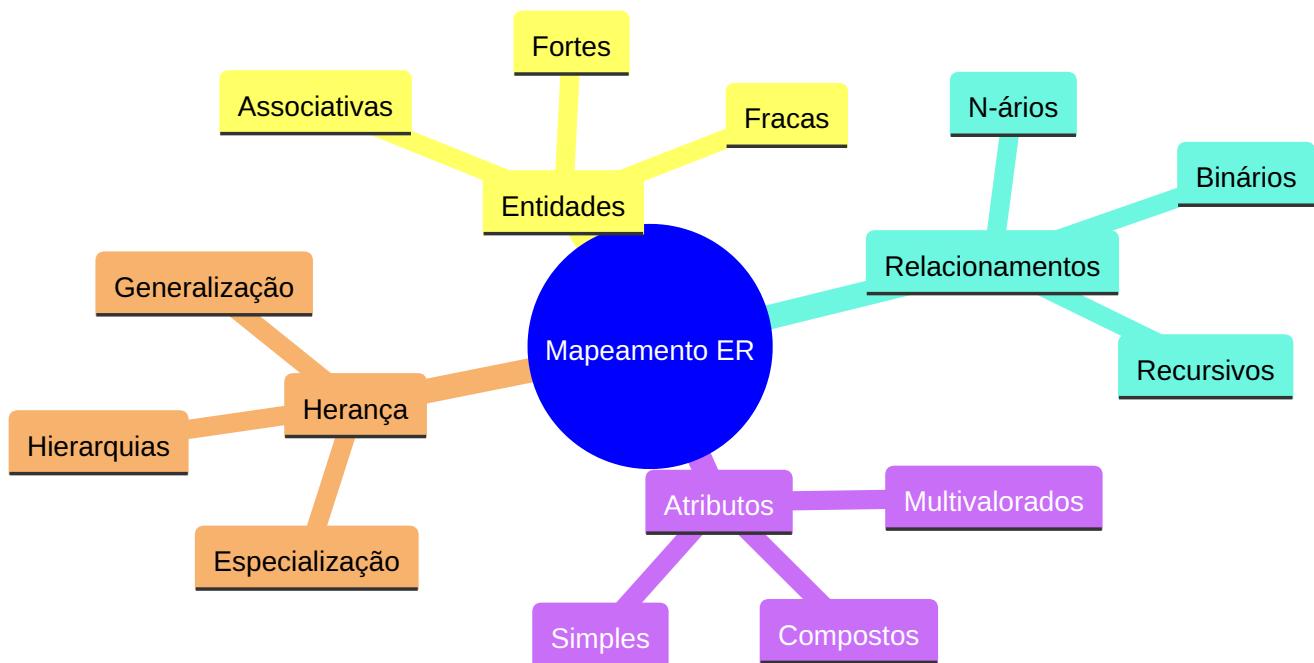
Benefícios da prevenção:

- Menos bugs
- Menor custo
- Maior satisfação
- Melhor evolução

Mapeamento do Modelo ER

O processo de mapeamento do Modelo Entidade-Relacionamento (ER) para um modelo lógico ou físico é uma etapa crucial no design de bancos de dados. Este capítulo aborda as estratégias e técnicas para realizar essa transformação de forma eficiente e consistente.

Visão Geral do Mapeamento



Princípios Fundamentais

1. Preservação Semântica

- Manter o significado dos dados
- Preservar regras de negócio
- Garantir integridade referencial
- Conservar restrições do modelo

2. Otimização Estrutural

- Minimizar redundância
- Otimizar acesso aos dados
- Balancear normalização
- Considerar performance

3. Consistência

- Seguir padrões de nomenclatura
- Manter convenções de design
- Documentar decisões
- Garantir rastreabilidade

Processo de Mapeamento

1. Análise do Modelo ER

2. Identificação de Estruturas

Entidades Fortes

- Mapeamento direto para tabelas
- Definição de chaves primárias
- Atributos como colunas
- Restrições de integridade

Relacionamentos

- Análise de cardinalidade
- Chaves estrangeiras
- Tabelas de associação

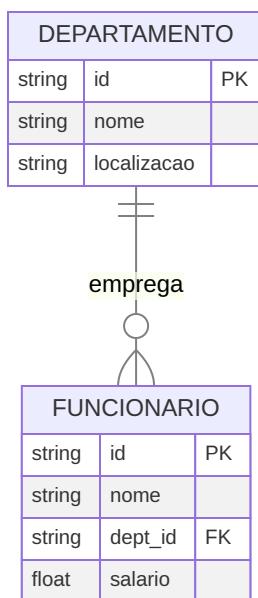
- Restrições de referência

Atributos Especiais

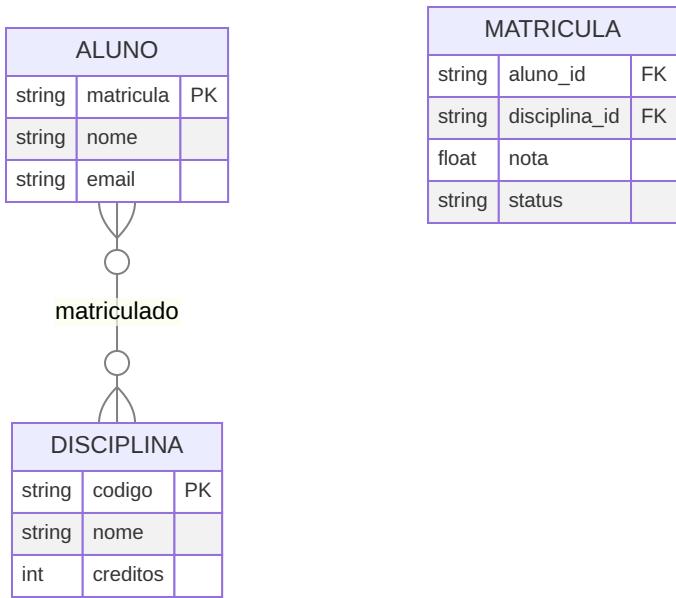
- Tratamento de compostos
- Resolução de multivalorados
- Derivados e calculados
- Tipos de dados apropriados

Exemplos de Transformação

1. Relacionamento 1:N



2. Relacionamento N:M



Considerações Práticas

1. Performance

- Índices apropriados
- Tipos de dados eficientes
- Estratégias de particionamento
- Otimização de consultas

2. Manutenibilidade

- Documentação clara
- Nomenclatura consistente
- Modularidade
- Flexibilidade para mudanças

3. Escalabilidade

- Estruturas extensíveis
- Gerenciamento de crescimento
- Estratégias de distribuição
- Planejamento de capacidade

Ferramentas e Tecnologias

1. Ferramentas CASE

- Modelagem visual
- Geração de código
- Documentação automática
- Validação de modelos

2. SGBDs

- Recursos específicos
- Limitações técnicas
- Otimizações disponíveis
- Extensões proprietárias

Próximos Passos

Para aprofundar seu conhecimento em mapeamento ER, explore:

1. Mapeamento de Entidades

- Tipos de entidades
- Atributos especiais
- Restrições específicas

2. Mapeamento de Relacionamentos

- Cardinalidades
- Participação
- Atributos de relacionamento

3. Mapeamento de Herança

- Estratégias de implementação
- Hierarquias
- Restrições específicas

Conclusão

O mapeamento ER é fundamental para:

- Implementação eficiente
- Integridade dos dados
- Performance do sistema
- Manutenibilidade do banco

Uma estratégia bem planejada garante:

- Consistência dos dados
- Facilidade de evolução
- Melhor desempenho
- Menor custo de manutenção

Mapeamento de Entidades

O processo de mapeamento de entidades é fundamental na transformação do modelo conceitual para o modelo relacional. Este processo requer compreensão profunda de teoria dos conjuntos, álgebra relacional e dependências funcionais.

Fundamentos Teóricos

1. Definições Básicas

Seja E uma entidade com atributos $A = \{A_1, A_2, \dots, A_n\}$:

- Domínio: $\text{dom}(A_i)$ é o conjunto de valores possíveis para A_i
- Tupla: $t \in \text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n)$
- Relação: $R \subseteq \text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n)$

2. Propriedades Fundamentais

1. Unicidade:

$$\forall t_1, t_2 \in R : t_1 \neq t_2 \rightarrow t_1[K] \neq t_2[K]$$

onde K é chave primária

2. Integridade Referencial:

$$\forall t_1 \in R_1 : t_1[FK] \neq \text{null} \rightarrow \exists t_2 \in R_2 : t_1[FK] = t_2[PK]$$

3. Dependência Funcional:

$$X \rightarrow Y \iff \forall t_1, t_2 \in R : t_1[X] = t_2[X] \rightarrow t_1[Y] = t_2[Y]$$

Regras de Mapeamento Detalhadas

1. Entidades Fortes

1.1 Definição Formal

Para uma entidade forte E(K, A₁, ..., A_n):

$$R = \{t \mid t \in \text{dom}(K) \times \text{dom}(A_1) \times \dots \times \text{dom}(A_n)\}$$

1.2 Propriedades

- Chave Primária: K → {A₁, ..., A_n}
- Unicidade: ∀t₁, t₂ ∈ R : t₁[K] = t₂[K] → t₁ = t₂
- Não-nulidade: ∀t ∈ R : t[K] ≠ null

1.3 Exemplo Matemático

```
PRODUTO = {
    (k, a1, ..., an) |
    k ∈ dom(código),
    a1 ∈ dom(nome),
    ...,
    an ∈ dom(atributon)
}
```

2. Entidades Fracas

2.1 Definição Formal

Para uma entidade fraca W dependente de E:

```
W = {(k1, fk, a1, ..., an) |
    k1 ∈ dom(discriminador),
    fk ∈ dom(E.K),
    ∃e ∈ E : e.K = fk
}
```

2.2 Dependências

- Chave Parcial: $k_1 \rightarrow_{\phi} \{a_1, \dots, a_n\}$
- Dependência Existencial: $W[fk] \subseteq E[K]$

2.3 Exemplo Detalhado

```
DEPENDENTE = {
    (seq, func_id, nome, data_nasc) |
    (seq, func_id) é único,
    func_id ∈ FUNCIONARIO[id]
}
```

3. Atributos Complexos

3.1 Atributos Compostos

Seja $C = \{c_1, \dots, c_n\}$ um atributo composto:

Método 1 - Decomposição:

```
R(K, c1, ..., cn, outros_atributos)
```

Método 2 - Nova Relação:

```
R(K, outros_atributos)
C(K, c1, ..., cn)
K → {c1, ..., cn}
```

3.2 Atributos Multivalorados

Para atributo multivalorado M:

```
E(K, A1, ..., An)
M(K, valor, metadata)
onde:
- K referencia E.K
- (K, valor) é único
```

Padrões de Mapeamento

1. Hierarquia de Generalização

1.1 Single Table

$R(K, A_1, \dots, A_n, B_1, \dots, B_m, C_1, \dots, C_k, \text{tipo})$

onde:

- A são atributos comuns
- B, C são atributos específicos
- tipo discrimina a subclasse

1.2 Table Per Class

$R_1(K, A_1, \dots, A_n)$

$R_2(K, A_1, \dots, A_n, B_1, \dots, B_m)$

$R_3(K, A_1, \dots, A_n, C_1, \dots, C_k)$

2. Agregação

Para agregação de E_1 e E_2 em AG:

$AG(K, FK_1, FK_2, A_1, \dots, A_n)$

onde:

- K é identificador próprio
- FK_1 referencia E_1
- FK_2 referencia E_2

Casos Especiais

1. Auto-relacionamento

$E(K, A_1, \dots, A_n)$

$R(K_1, K_2, \text{tipo_relação})$

onde:

- K_1, K_2 referenciam $E.K$
- (K_1, K_2) é único

2. Relacionamentos n-ários

Para relacionamento entre E_1, \dots, E_n :

$R(K_1, \dots, K_n, A_1, \dots, A_m)$

onde:

- K_i referencia $E_i.K$
- (K_1, \dots, K_n) é chave primária

Otimizações e Considerações

1. Análise de Dependências

1.1 Dependências Funcionais

$$F^+ = \{X \rightarrow Y \mid X \rightarrow Y \text{ é derivável de } F\}$$

1.2 Dependências Multivaloradas

$X \rightarrow\! \rightarrow Y$ significa que Y depende multivalorado de X

2. Normalização

2.1 Primeira Forma Normal (1NF)

- Atomicidade dos atributos
- Não permite grupos repetitivos

2.2 Segunda Forma Normal (2NF)

- Satisfaz 1NF
- Não há dependências parciais

2.3 Terceira Forma Normal (3NF)

- Satisfaz 2NF
- Não há dependências transitivas

Exemplos Avançados

1. Sistema de Recursos Humanos

```
FUNCIONARIO(id, nome, data_admissao)
CARGO(código, nome, nível)
HISTÓRICO_CARGO(func_id, cargo_id, data_início, data_fim)
HABILIDADE(func_id, tipo, nível, certificação)
```

2. Sistema Acadêmico Completo

```
ALUNO(matrícula, nome, curso)
DISCIPLINA(código, nome, créditos)
PROFESSOR(id, nome, departamento)
TURMA(id, disciplina_id, professor_id, semestre)
MATRÍCULA(aluno_id, turma_id, nota, frequência)
```

Validação do Mapeamento

1. Critérios de Qualidade

1. Preservação de Informação:

$$\forall e \in E, \exists t \in R : \text{representa}(t, e)$$

2. Preservação de Dependências:

$$\forall d \in D, \text{mapeamento}(d) \in D'$$

onde D, D' são conjuntos de dependências

3. Minimização de Redundância:

$\nexists t_1, t_2 \in R : \text{duplica_info}(t_1, t_2)$

2. Testes de Integridade

1. Teste de Chaves:

$\forall R, \exists K : K \text{ é minimal e } K \rightarrow R$

2. Teste de Referências:

$\forall FK \in R_1, \exists PK \in R_2 : FK \subseteq PK$

Conclusão

O mapeamento efetivo requer:

1. Compreensão profunda da teoria
2. Aplicação consistente das regras
3. Consideração dos requisitos específicos
4. Validação rigorosa do resultado

Pontos críticos:

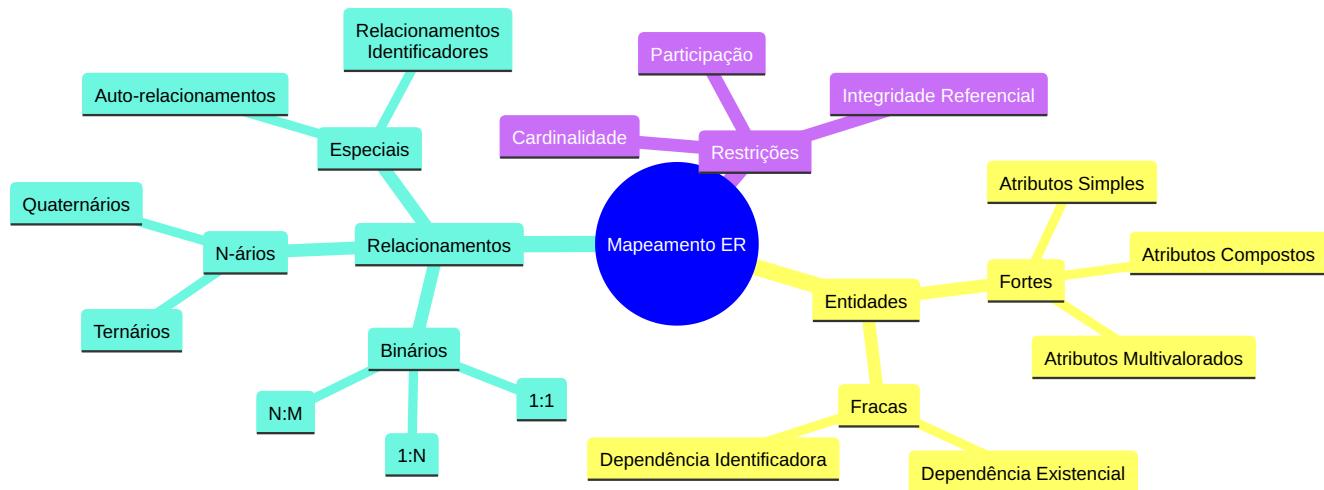
- Preservação semântica
- Integridade referencial
- Normalização adequada
- Eficiência operacional

Mapeamento do Modelo ER

O mapeamento do Modelo Entidade-Relacionamento (ER) para o modelo relacional é um processo sistemático que requer compreensão profunda de ambos os modelos. Este capítulo apresenta uma abordagem estruturada para realizar essa transformação.

Fundamentos do Mapeamento

Conceitos Básicos



Princípios Fundamentais

1. Preservação de Informação

- Manutenção de todos os dados
- Conservação das relações
- Integridade dos atributos

2. Garantia de Consistência

- Restrições de integridade
- Regras de negócio
- Validações estruturais

3. Otimização de Acesso

- Eficiência nas consultas
- Minimização de junções
- Estruturas de índice

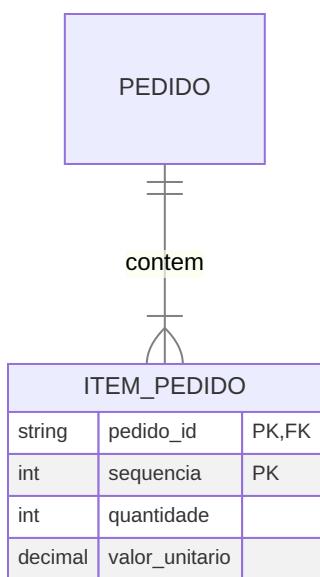
Mapeamento de Entidades

1. Entidades Fortes

Regras de Transformação

- Cada entidade forte torna-se uma tabela
- Atributos tornam-se colunas
- Chave primária é preservada
- Restrições são mapeadas para constraints

2. Entidades Fracas



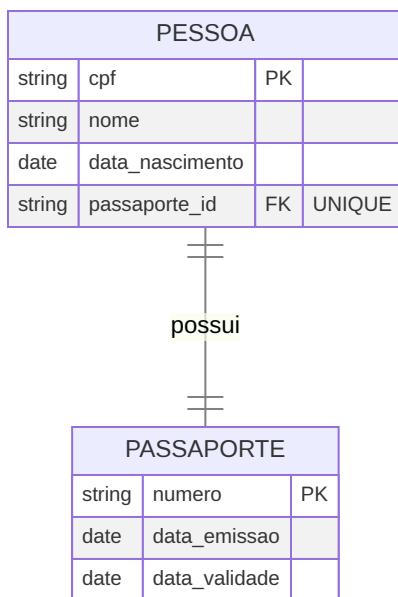
Considerações Especiais

- Dependência da entidade forte

- Chave parcial como parte da PK
- Restrições de integridade referencial

Mapeamento de Relacionamentos

1. Relacionamentos 1:1



Estratégias de Implementação

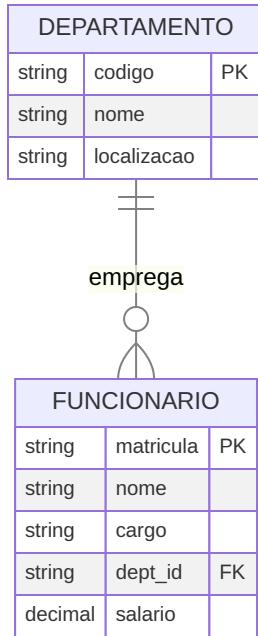
1. Participação Total-Total

- Chave estrangeira em qualquer lado
- Restrição UNIQUE na FK
- Restrição NOT NULL

2. Participação Total-Parcial

- FK no lado total
- Restrição UNIQUE
- Permite NULL no lado parcial

2. Relacionamentos 1:N



Regras de Mapeamento

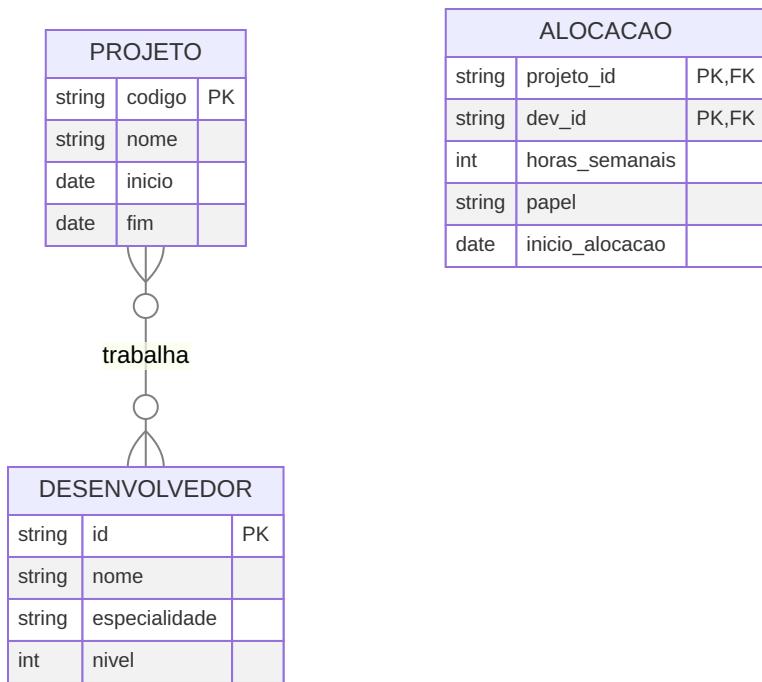
1. Participação Total

- FK NOT NULL no lado N
- Índice na FK
- Trigger para validação

2. Participação Parcial

- FK permite NULL
- Constraints específicas
- Índices seletivos

3. Relacionamentos N:M



Técnicas de Implementação

1. Tabela de Associação

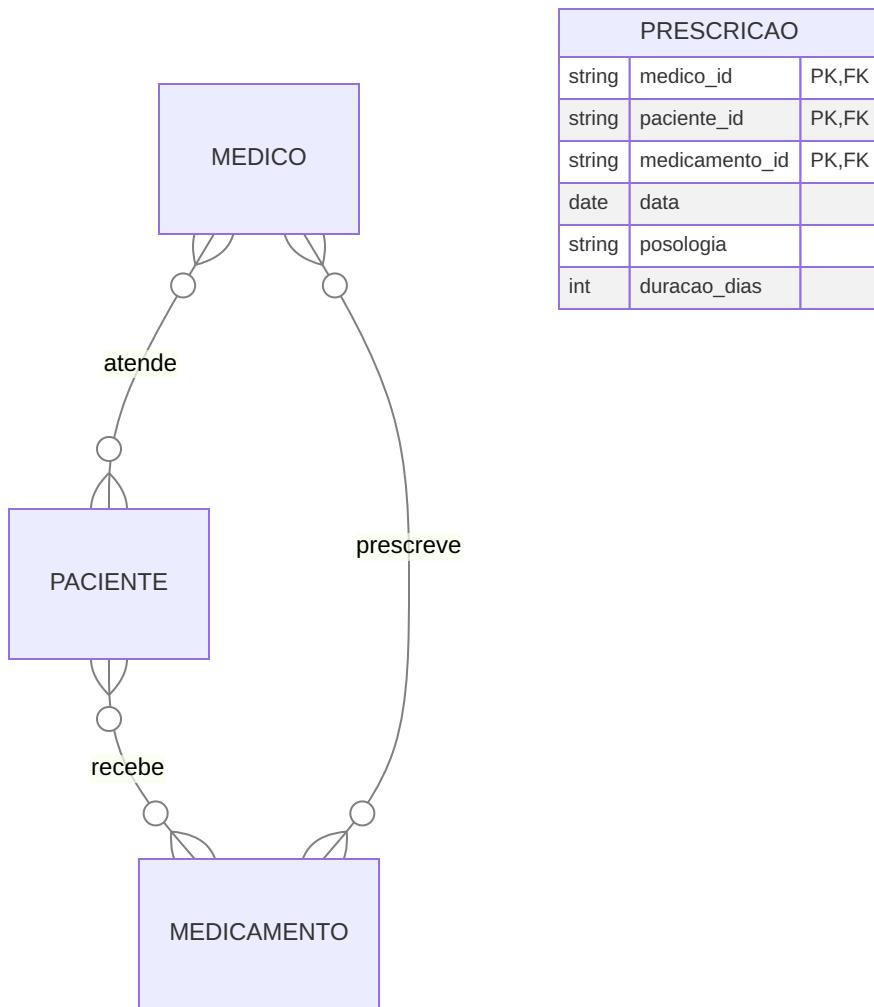
- Chave composta
- Atributos do relacionamento
- Índices compostos

2. Otimizações

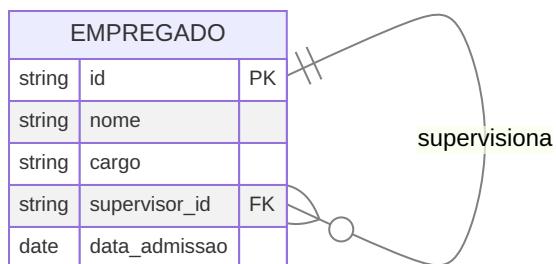
- Índices específicos
- Particionamento
- Clustering

Casos Especiais

1. Relacionamentos Ternários



2. Auto-relacionamentos



Otimizações e Considerações

1. Performance

Estruturas de Índice

- Índices primários
- Índices secundários
- Índices compostos
- Índices parciais

Estratégias de Junção

- Inner joins otimizados
- Outer joins seletivos
- Semi-joins quando aplicável

2. Integridade

Restrições de Domínio

- CHECK constraints
- DEFAULT values
- Triggers de validação

Integridade Referencial

- ON DELETE actions
- ON UPDATE actions
- Deferrable constraints

Validação do Esquema

1. Testes de Integridade

Verificações Básicas

- Chaves primárias
- Chaves estrangeiras

- Unicidade
- NOT NULL constraints

Testes Avançados

- Cardinalidade
- Participação
- Regras de negócio

2. Análise de Qualidade

Métricas

- Normalização
- Redundância
- Complexidade
- Performance

Otimizações

- Desnormalização seletiva
- Índices compostos
- Particionamento
- Clustering

Conclusão

O mapeamento efetivo do modelo ER requer:

1. Compreensão Profunda

- Modelo conceitual

- Modelo relacional
- Requisitos do sistema

2. Abordagem Sistemática

- Metodologia clara
- Documentação adequada
- Validação rigorosa

3. Considerações Práticas

- Performance
- Manutenibilidade
- Escalabilidade
- Evolução futura

Próximos Passos

1. Implementação Física

- Escolha do SGBD
- Scripts de criação
- Migração de dados

2. Monitoramento

- Performance
- Integridade
- Uso do sistema

3. Manutenção

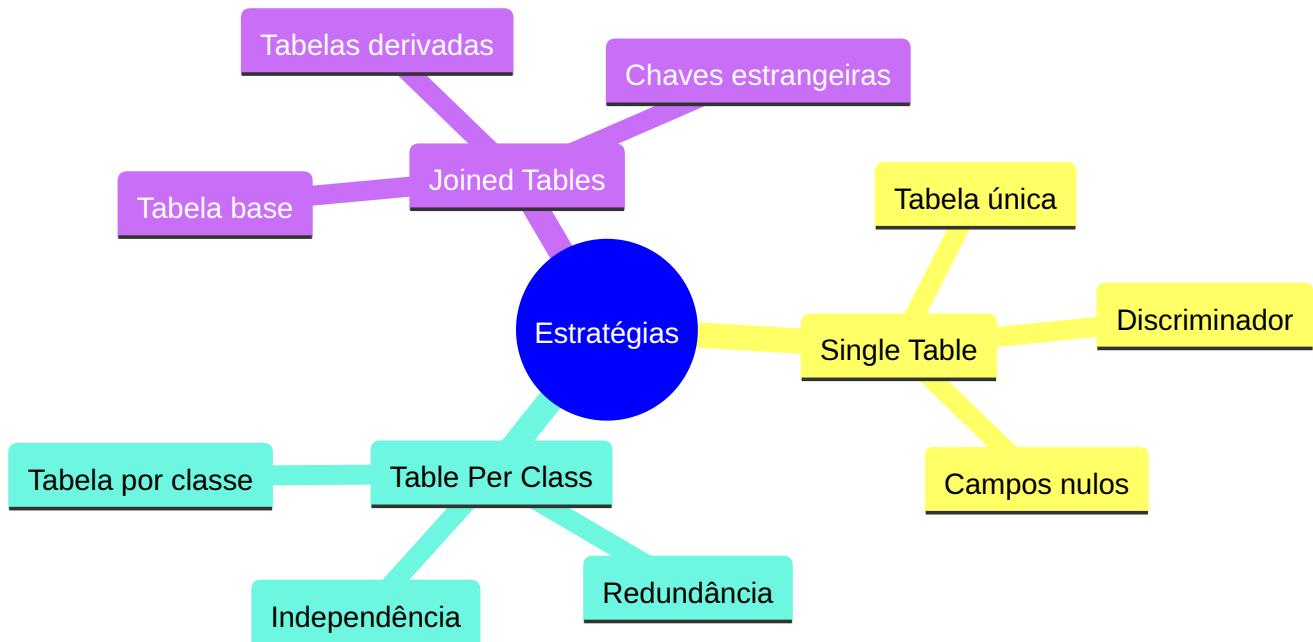
- Ajustes de esquema

- Otimizações
- Documentação

Mapeamento de Herança

O mapeamento de hierarquias de herança do modelo ER para o modelo relacional requer estratégias específicas para preservar a semântica e garantir eficiência. Este capítulo apresenta as principais abordagens e suas implicações.

Visão Geral das Estratégias



Estratégias de Mapeamento

1. Single Table (Tabela Única)

Exemplo: Sistema Acadêmico

PESSOA			
		PK	
string	id		
string	nome		
string	tipo		
string	matricula	Aluno	
string	siape	Professor	
string	departamento	Professor	
float	cr	Aluno	
string	turma	Aluno	
string	titulacao	Professor	
string	sala	Professor	

Exemplo: Sistema de Veículos

VEICULO			
		PK	
string	placa		
string	modelo		
string	tipo		
int	num_portas	Carro	
float	capacidade_carga	Caminhao	
int	num_eixos	Caminhao	
int	cilindradas	Moto	
string	tipo_guidao	Moto	

Características

- Todos os atributos em uma única tabela
- Campo discriminador para identificar subclasses
- Atributos específicos podem ser nulos

Vantagens

- Consultas simples
- Sem necessidade de junções
- Fácil manutenção

Desvantagens

- Desperdício de espaço
- Muitos campos nulos
- Menor integridade de dados

2. Table Per Class (Tabela por Classe)

Exemplo: Sistema Acadêmico

PESSOA		
string	id	PK
string	nome	
string	email	

ALUNO		
string	id	PK
string	nome	
string	email	
string	matricula	
float	cr	
string	turma	

PROFESSOR		
string	id	PK
string	nome	
string	email	
string	siape	
string	departamento	
string	titulacao	
string	sala	

Exemplo: Sistema de Produtos

PRODUTO		
string	codigo	PK
string	nome	
float	preco	

ELETRONICO		
string	codigo	PK
string	nome	
float	preco	
string	voltagem	
string	garantia	
float	potencia	

LIVRO		
string	codigo	PK
string	nome	
float	preco	
string	isbn	
string	autor	
int	paginas	

ALIMENTO		
string	codigo	PK
string	nome	
float	preco	
date	validade	
float	peso	
string	nutricional	

Características

- Cada classe tem sua própria tabela
- Todos os atributos são replicados
- Chaves independentes

Vantagens

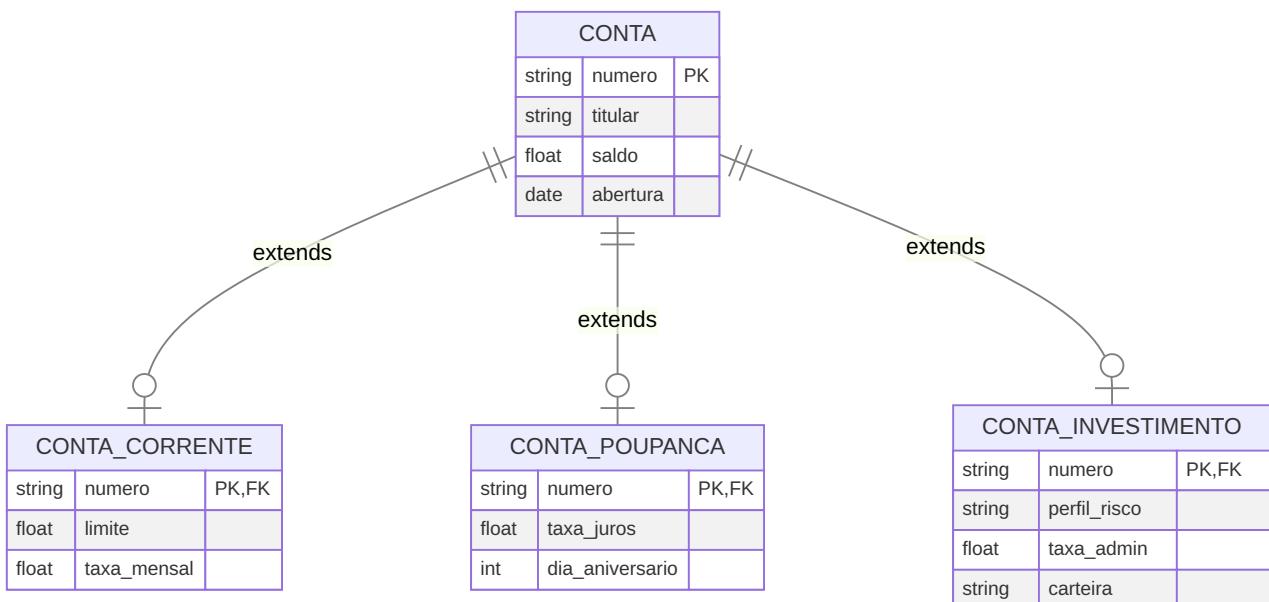
- Modelo mais flexível
- Melhor integridade de dados
- Consultas específicas eficientes

Desvantagens

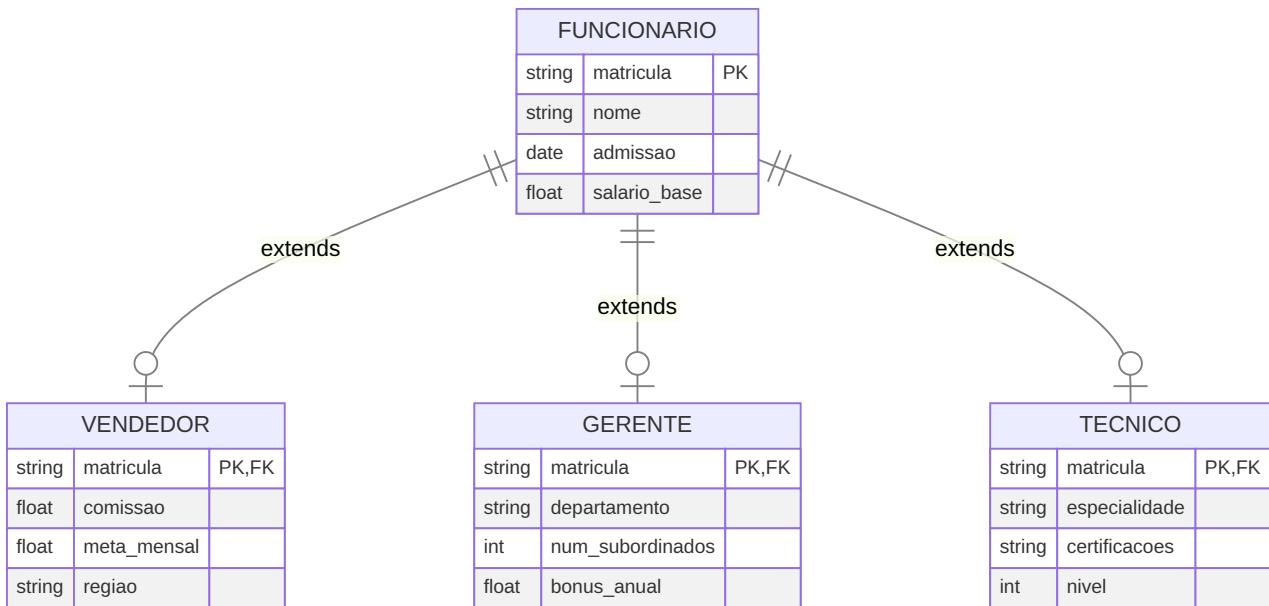
- Redundância de dados
- Consultas polimórficas complexas
- Maior espaço de armazenamento

3. Joined Tables (Tabelas Unidas)

Exemplo: Sistema Bancário



Exemplo: Sistema de Funcionários



Características

- Tabela base para superclasse
- Tabelas separadas para subclasses
- Chaves estrangeiras para relacionamento

Vantagens

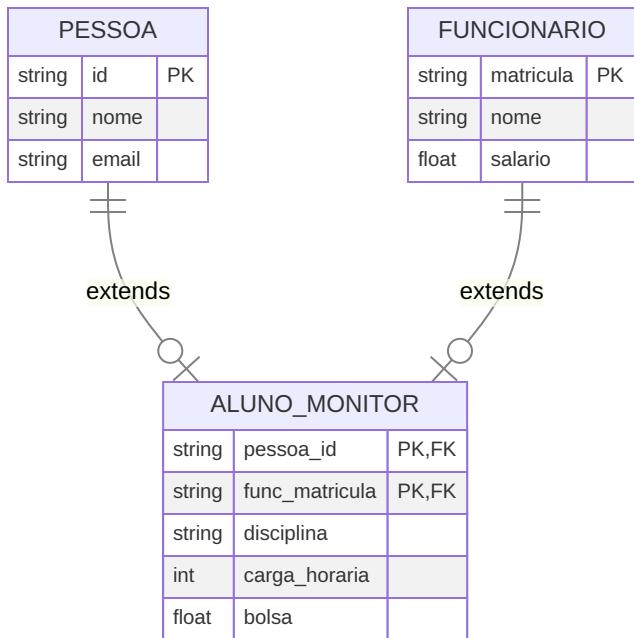
- Normalização completa
- Sem campos nulos
- Integridade referencial

Desvantagens

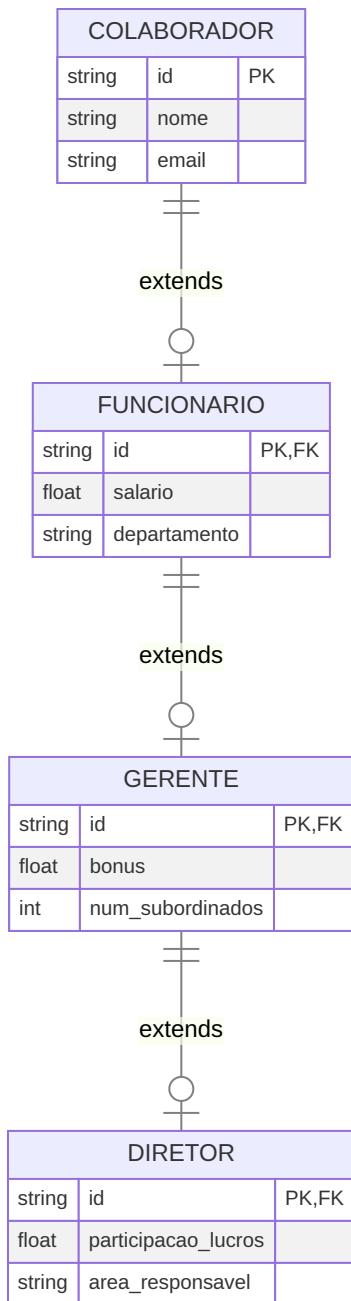
- Necessidade de junções
- Performance reduzida
- Complexidade de manutenção

Casos Especiais

1. Herança Múltipla

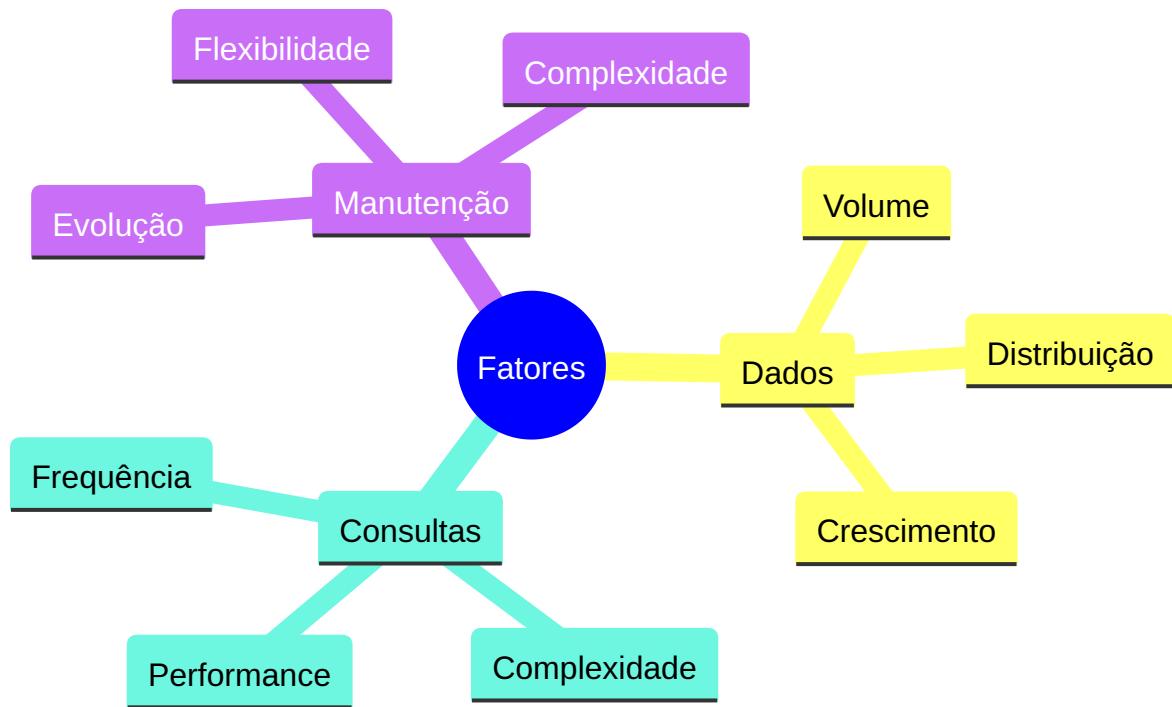


2. Herança Hierárquica



Considerações de Design

1. Escolha da Estratégia



Fatores de Decisão

- Volume de dados
- Padrão de consultas
- Requisitos de integridade
- Flexibilidade necessária

Recomendações

1. Single Table

- Hierarquias simples
- Poucos atributos específicos
- Consultas frequentes polimórficas

2. Table Per Class

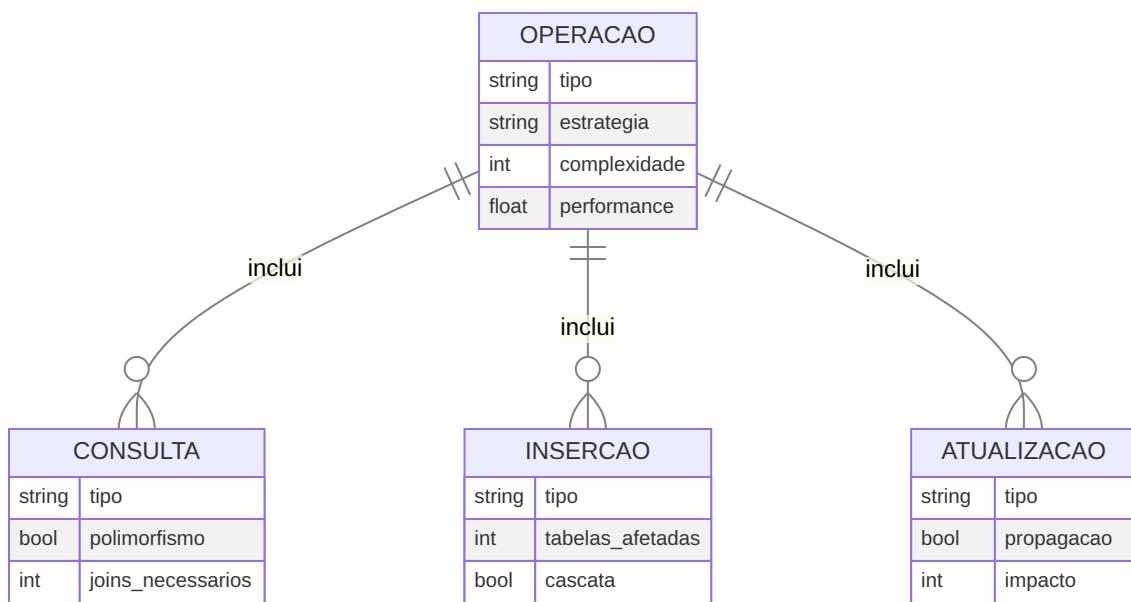
- Subclasses muito diferentes
- Consultas específicas frequentes

- Poucos dados compartilhados

3. Joined Tables

- Alta normalização necessária
- Dados compartilhados importantes
- Evolução frequente do esquema

2. Impacto nas Operações



2. Otimizações

Índices

- Chaves primárias
- Campos discriminadores
- Chaves estrangeiras
- Campos de consulta frequente

Restrições

- Integridade referencial

- Validações de tipo
- Regras de negócio

Padrões e Anti-padrões

Padrões Recomendados

1. Discriminador Explícito

- Campo tipo sempre presente
- Validações consistentes
- Documentação clara

2. Nomenclatura Consistente

- Prefixos/sufixos padronizados
- Relacionamentos claros
- Convenções estabelecidas

Anti-padrões

1. Mistura de Estratégias

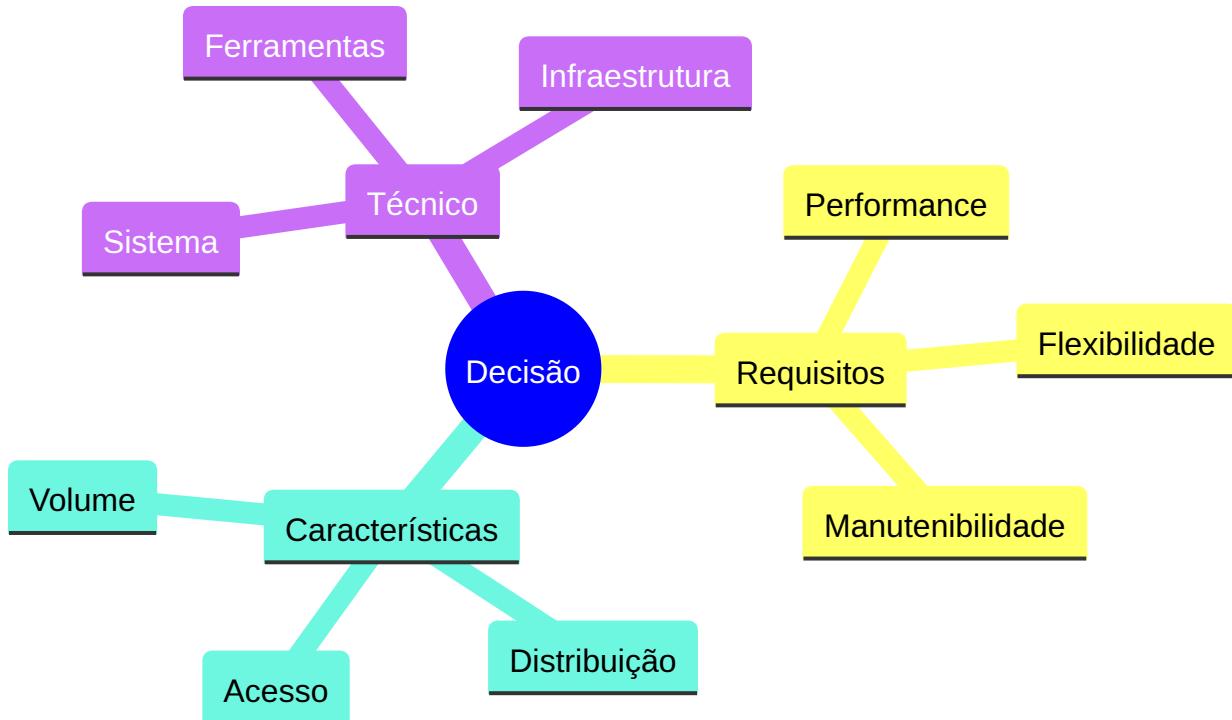
- Inconsistência no modelo
- Complexidade desnecessária
- Difícil manutenção

2. Herança Profunda

- Muitos níveis hierárquicos
- Performance degradada
- Complexidade aumentada

Conclusão

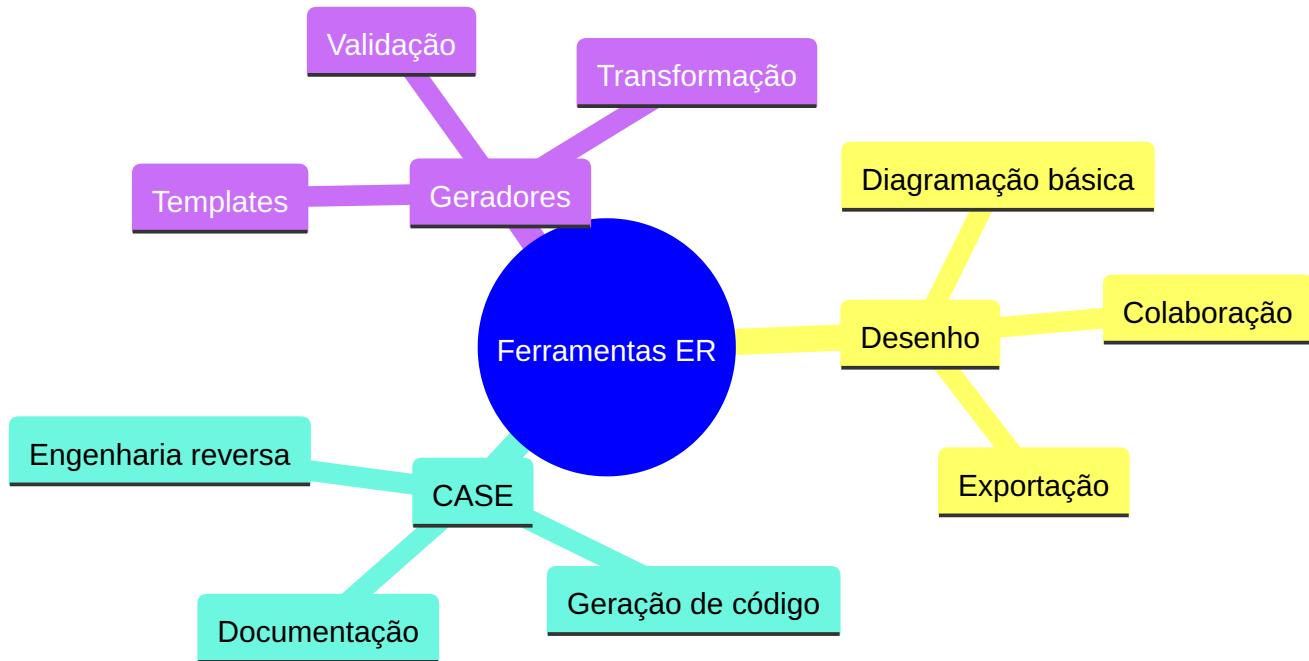
A escolha da estratégia de mapeamento de herança deve considerar:



Ferramentas para Modelagem ER

As ferramentas de modelagem Entidade-Relacionamento são essenciais para criar, manter e documentar modelos de dados de forma eficiente.

Categorias Principais



Critérios de Seleção

1. Aspectos Técnicos

- Suporte a notações (Chen, Crow's Foot, UML)
- Capacidade de exportação
- Integração com outros sistemas
- Validação de modelos

2. Aspectos Práticos

- Curva de aprendizado

- Custo x benefício
- Suporte e comunidade
- Atualizações regulares

Recomendações Gerais

1. Para Iniciantes

- Ferramentas online gratuitas
- Interface intuitiva
- Documentação em português
- Comunidade ativa

2. Para Profissionais

- Ferramentas enterprise
- Recursos avançados
- Integração com IDEs
- Suporte empresarial

3. Para Educação

- Ferramentas didáticas
- Exemplos práticos
- Material de apoio
- Licenças acadêmicas

Ferramentas de Desenho

Ferramentas Populares

1. Draw.io

- Gratuito e open-source
- Interface web e desktop
- Múltiplas notações
- Exportação versátil

2. Lucidchart

- Colaboração em tempo real
- Templates profissionais
- Integração com cloud
- Controle de versão

3. Microsoft Visio

- Padrão empresarial
- Integração Office
- Templates extensivos
- Recursos avançados

Recursos Essenciais

Básicos

- Formas padrão ER
- Conectores inteligentes
- Grade e alinhamento
- Zoom e navegação

Avançados

- Validação de diagramas
- Versionamento
- Colaboração
- Exportação múltipla

Melhores Práticas

1. Organização

- Use camadas
- Agrupe elementos
- Mantenha alinhamento
- Padronize cores

2. Produtividade

- Aprenda atalhos
- Use templates
- Backup regular
- Versione trabalho

3. Colaboração

- Documente decisões

- Compartilhe templates
- Estabeleça padrões
- Revise em equipe

Ferramentas CASE

Principais Soluções

1. Enterprise Architect

- Modelagem completa
- Engenharia reversa
- Documentação detalhada
- Suporte multiplataforma

2. PowerDesigner

- Padrão industrial
- Modelagem múltipla
- Repositório central
- Análise de impacto

3. ERwin Data Modeler

- Foco em dados
- Modelagem dimensional
- Comparação de modelos
- Governança de dados

Funcionalidades Chave

Modelagem

- Diagramas ER
- Modelos lógicos
- Modelos físicos
- Normalização

Engenharia

- Forward engineering
- Reverse engineering
- Round-trip engineering
- Sincronização

Documentação

- Geração automática
- Relatórios customizados
- Dicionário de dados
- Metadados

Metodologia de Uso

1. Planejamento

- Definir padrões
- Configurar ambiente
- Treinar equipe
- Estabelecer workflow

2. Implementação

- Criar repositório
- Importar modelos
- Configurar segurança
- Definir processos

3. Manutenção

- Backup regular
- Atualizar modelos
- Revisar permissões
- Auditar uso

Geradores de Código

Tipos de Geradores

1. DDL Generators

- Scripts SQL
- Constraints
- Índices
- Procedures

2. ORM Generators

- Classes de domínio
- Mapeamentos
- Configurações
- Migrations

3. Documentation Generators

- Documentação técnica
- Dicionário de dados
- Diagramas
- Relatórios

Recursos Avançados

Customização

- Templates
- Convenções
- Nomenclatura
- Padrões

Validação

- Regras de negócio
- Constraints
- Relacionamentos
- Integridade

Versionamento

- Controle de mudanças
- Histórico
- Rollback
- Migrations

Boas Práticas

1. Configuração

- Defina padrões claros
- Configure templates
- Estabeleça convenções
- Documente decisões

2. Uso

- Valide antes de gerar
- Revise código gerado
- Mantenha consistência
- Versione artefatos

3. Manutenção

- Atualize templates
- Refine regras
- Monitore qualidade
- Colete feedback

Ferramentas Populares

Database-First

- Schema Spy
- MySQL Workbench
- pgModeler
- Oracle SQL Developer

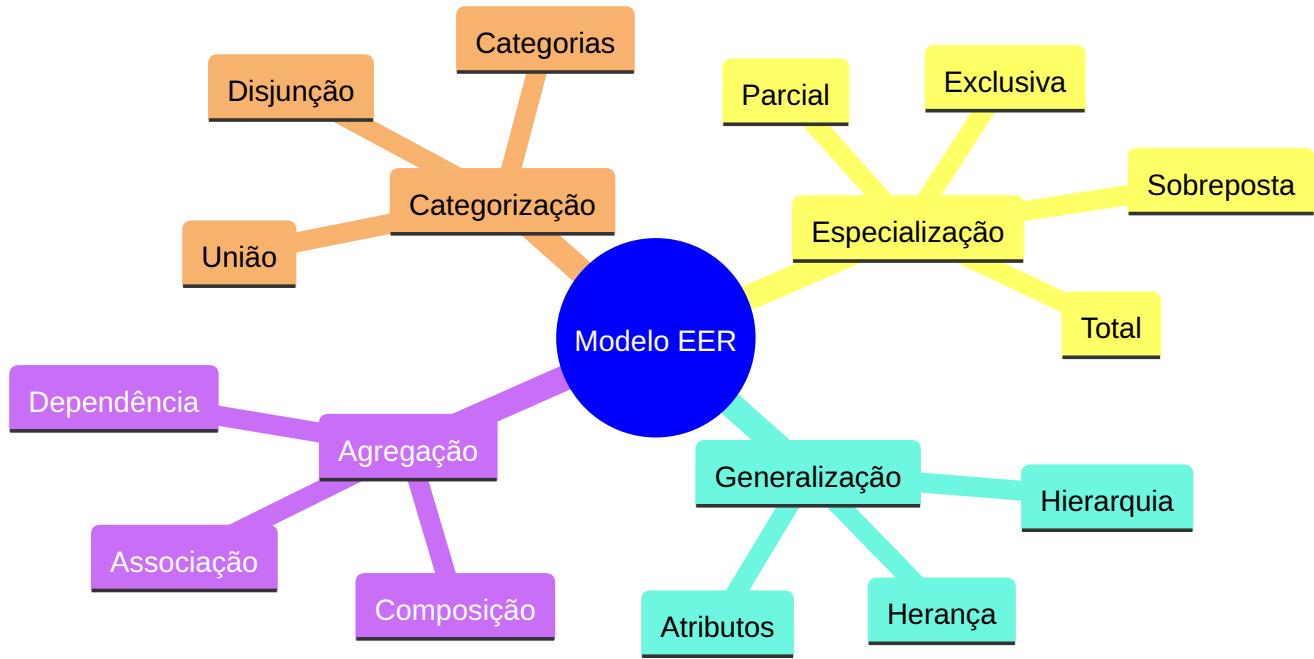
Code-First

- Entity Framework
- Hibernate Tools
- JOOQ
- TypeORM

Modelo ER Estendido (EER)

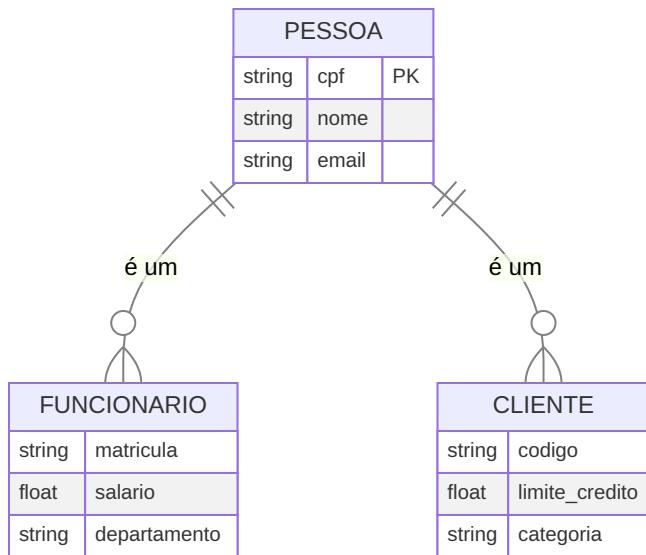
O Modelo ER Estendido adiciona conceitos avançados ao modelo ER tradicional, permitindo uma modelagem mais rica e precisa.

Conceitos Principais

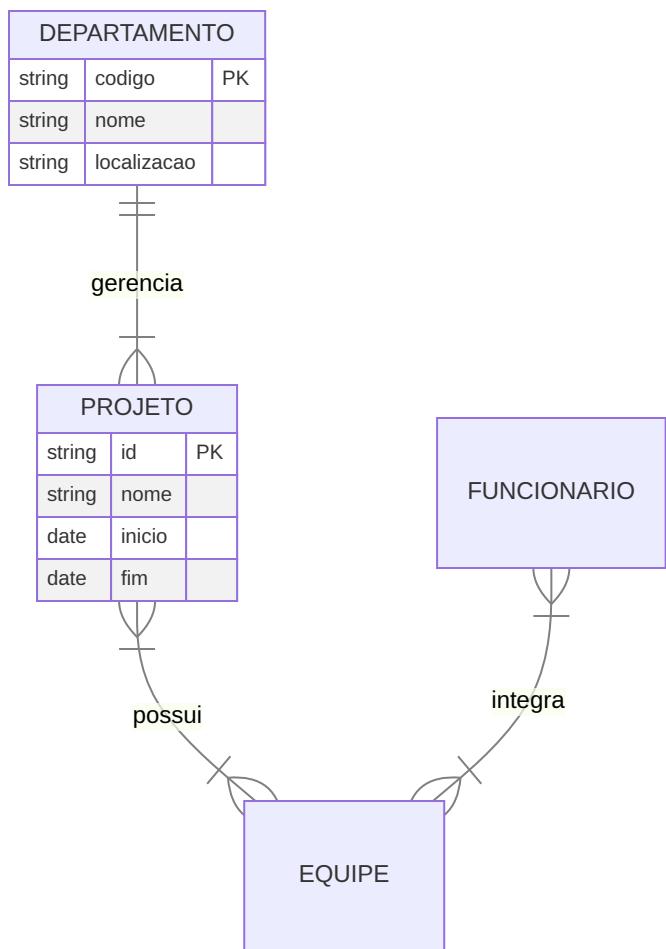


Elementos Avançados

1. Especialização/Generalização



2. Agregação e Composição



Aplicações Práticas

1. Modelagem de Herança

- Hierarquias naturais
- Compartilhamento de atributos
- Especialização de comportamentos
- Restrições de integridade

2. Modelagem de Composição

- Relacionamentos todo-parte
- Dependências existenciais
- Propagação de operações
- Integridade referencial

3. Modelagem de Categorização

- Tipos dinâmicos
- União de entidades
- Restrições de participação
- Regras de negócio

Vantagens e Limitações

Vantagens

- Maior expressividade
- Melhor semântica

- Reutilização
- Organização

Limitações

- Complexidade adicional
- Mapeamento mais difícil
- Implementação complexa
- Overhead de design

Melhores Práticas

1. Design

- Use quando necessário
- Mantenha simplicidade
- Documente decisões
- Valide com stakeholders

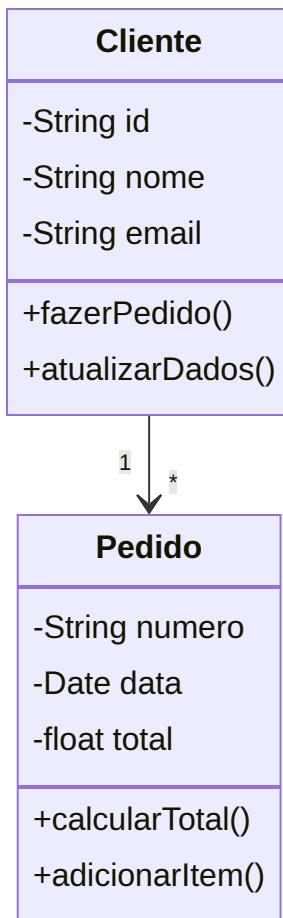
2. Implementação

- Planeje mapeamento
- Considere performance
- Teste integridade
- Monitore complexidade

Diagramas de Classe UML

Os diagramas de classe UML são fundamentais para modelagem orientada a objetos e podem ser usados como alternativa ou complemento aos modelos ER.

Elementos Básicos



Tipos de Relacionamentos

1. Associações

- Unidirecional
- Bidirecional
- Multiplicidade

- Papéis

2. Herança

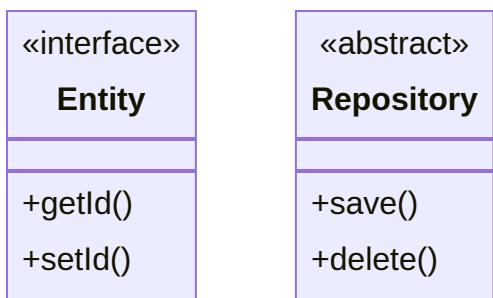
- Generalização
- Especialização
- Abstração
- Polimorfismo

3. Dependências

- Uso
- Criação
- Implementação
- Realização

Modelagem Avançada

1. Estereótipos



2. Restrições

- Invariantes
- Pré-condições
- Pós-condições

- Regras de negócio

3. Padrões

- Singleton
- Factory
- Observer
- Strategy

Comparação com ER

Semelhanças

- Modelagem estrutural
- Relacionamentos
- Atributos
- Restrições

Diferenças

- Foco em comportamento
- Herança nativa
- Interfaces
- Métodos

Melhores Práticas

1. Design

- Coesão alta

- Acoplamento baixo
- Encapsulamento
- Abstração adequada

2. Documentação

- Nomes claros
- Visibilidade correta
- Relacionamentos precisos
- Cardinalidade explícita

3. Manutenção

- Versionamento
- Refatoração
- Revisão
- Atualização

Ferramentas de Suporte

1. Modelagem

- Enterprise Architect
- StarUML
- Visual Paradigm
- Lucidchart

2. Geração de Código

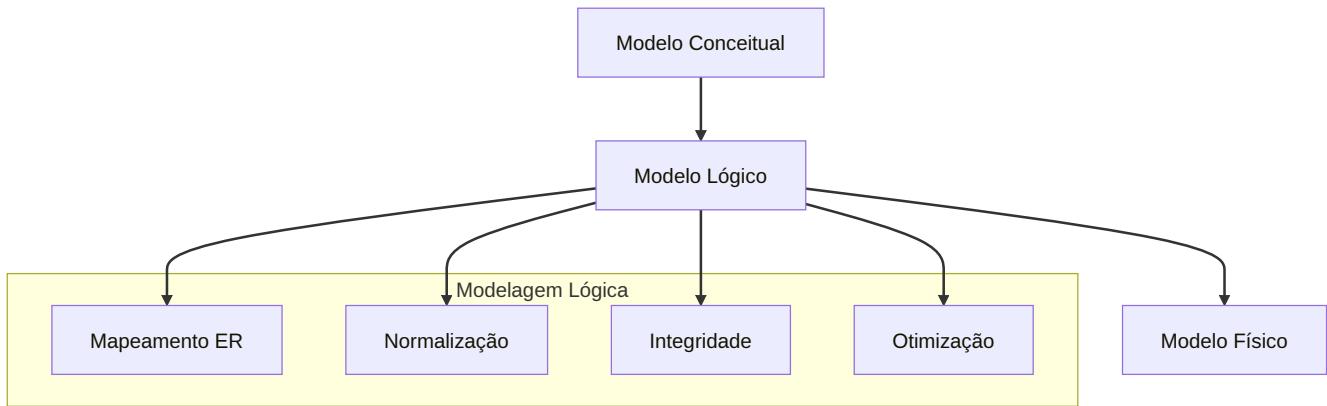
- Forward engineering

- Reverse engineering
- Round-trip engineering
- Templates

Modelagem Lógica

A modelagem lógica representa a segunda fase do processo de modelagem de dados, transformando o modelo conceitual em uma estrutura mais próxima da implementação.

Visão Geral



Processo de Transformação

1. Mapeamento ER para Relacional

Regras de Mapeamento

1. Entidades

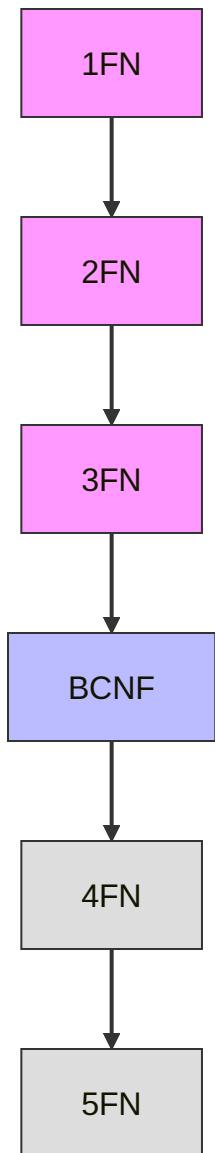
- Cada entidade vira uma tabela
- Atributos viram colunas
- Identificadores viram chaves primárias

2. Relacionamentos

- 1:1 → Chave estrangeira
- 1:N → Chave estrangeira no lado N
- N:M → Tabela associativa

Normalização

Formas Normais



Exemplo de Normalização

Tabela Original

```
PEDIDO (numero_pedido, data, cliente_nome, cliente_email,  
produto_nome, quantidade, preco_unitario)
```

Após Normalização

```

CREATE TABLE Cliente (
    id INT PRIMARY KEY,
    nome VARCHAR(100),
    email VARCHAR(100)
);

CREATE TABLE Pedido (
    numero INT PRIMARY KEY,
    data DATE,
    cliente_id INT REFERENCES Cliente(id)
);

CREATE TABLE ItemPedido (
    pedido_numero INT,
    produto_id INT,
    quantidade INT,
    preco_unitario DECIMAL(10, 2),
    PRIMARY KEY (pedido_numero, produto_id)
);

```

Integridade de Dados

1. Restrições de Integridade

- **Entidade**
 - Chaves primárias
 - Valores únicos
 - Não nulos
- **Referencial**
 - Chaves estrangeiras
 - Ações referenciais
 - Consistência

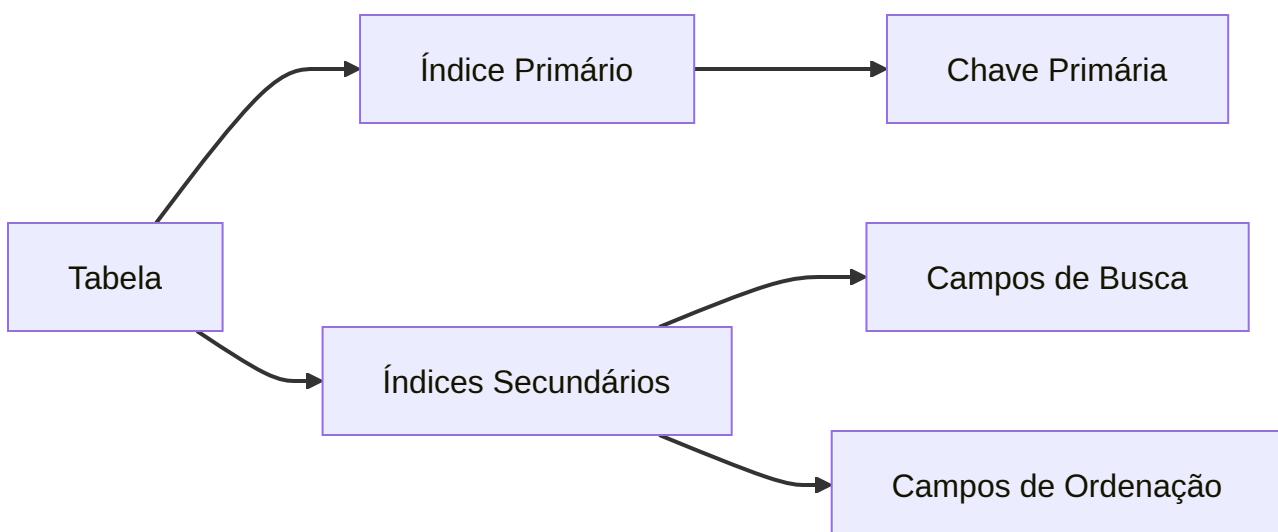
- Domínio
 - Tipos de dados
 - Intervalos válidos
 - Regras de negócio

2. Exemplo de Restrições

```
CREATE TABLE Produto (
    id INT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    preco DECIMAL(10,2) CHECK (preco > 0),
    categoria_id INT,
    FOREIGN KEY (categoria_id)
        REFERENCES Categoria(id)
        ON DELETE RESTRICT
        ON UPDATE CASCADE
);
```

Otimização Lógica

1. Índices



2. Visões

```
CREATE VIEW PedidosCliente AS
SELECT
    c.nome,
    COUNT(p.numero) as total_pedidos,
    SUM(p.valor) as valor_total
FROM
    Cliente c
    LEFT JOIN Pedido p ON c.id = p.cliente_id
GROUP BY
    c.id, c.nome;
```

Considerações de Design

1. Performance

- Estrutura de tabelas
- Relacionamentos
- Índices
- Particionamento

2. Manutenibilidade

- Nomenclatura
- Documentação
- Versionamento
- Padrões

3. Escalabilidade

- Distribuição
- Replicação

- Particionamento
- Cache

Ferramentas e Tecnologias

1. Modelagem

- MySQL Workbench
- Oracle SQL Developer
- ERwin
- PowerDesigner

2. Documentação

- Markdown
- PlantUML
- Mermaid
- Draw.io

Melhores Práticas

1. Nomenclatura

- Padrões consistentes
- Nomes significativos
- Prefixos/sufixos
- Documentação

2. Estruturação

- Normalização adequada

- Índices eficientes
- Relacionamentos claros
- Integridade garantida

3. Validação

- Testes de integridade
- Verificação de performance
- Revisão por pares
- Prova de conceito

Modelo Relacional

O modelo relacional é um modelo de dados que organiza as informações em relações (tabelas), baseado em conceitos matemáticos da teoria dos conjuntos e lógica de predicados.

Conceitos Fundamentais

1. Relação (Tabela)

FUNCIONARIO		
int	id	PK
string	nome	
string	cargo	
decimal	salario	
int	depto_id	FK

Uma relação consiste em:

- **Esquema:** estrutura da tabela
- **Tuplas:** linhas da tabela
- **Atributos:** colunas da tabela
- **Domínios:** tipos de dados válidos

2. Chaves

Tipos de Chaves

- **Chave Primária:** Identifica unicamente cada tupla
- **Chave Candidata:** Potencial chave primária
- **Chave Estrangeira:** Referencia chave primária de outra relação
- **Chave Composta:** Formada por múltiplos atributos

```

CREATE TABLE Pedido (
    numero INT,
    item_id INT,
    quantidade INT,
    PRIMARY KEY (numero, item_id),
    FOREIGN KEY (item_id) REFERENCES Item(id)
);

```

Propriedades Fundamentais

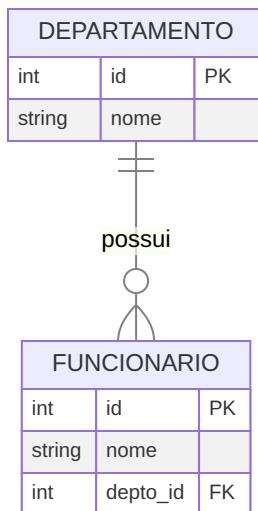
1. Integridade

Integridade de Entidade

- Chave primária não pode ser nula
- Valores únicos para identificação

Integridade Referencial

- Chaves estrangeiras válidas
- Consistência entre relações

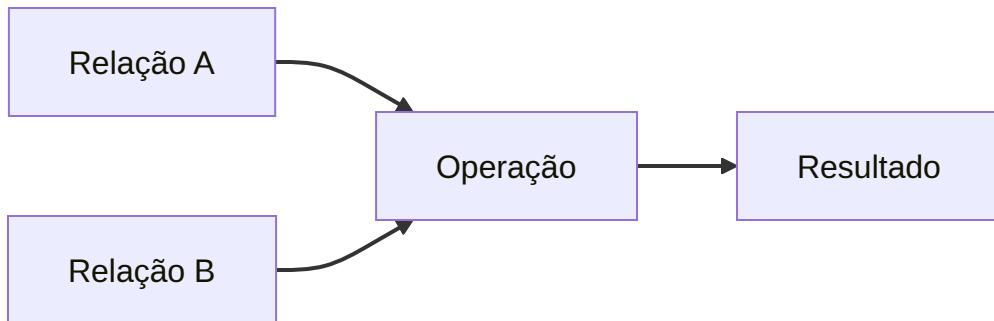


2. Atomicidade

- Valores atômicos (indivisíveis)
- Sem atributos multivalorados
- Sem grupos repetitivos

Operações Relacionais

1. Álgebra Relacional



Operações Básicas

- **Seleção (σ)**: Filtra tuplas
- **Projeção (π)**: Seleciona colunas
- **união (\cup)**: Combina tuplas
- **Diferença (-)**: Remove tuplas
- **Produto Cartesiano (\times)**: Combina todas as tuplas

Operações Derivadas

- **Interseção (\cap)**
- **Junção (\bowtie)**
- **Divisão (\div)**

2. Exemplos Práticos

```

-- Seleção
SELECT * FROM Funcionario
WHERE salario > 5000;

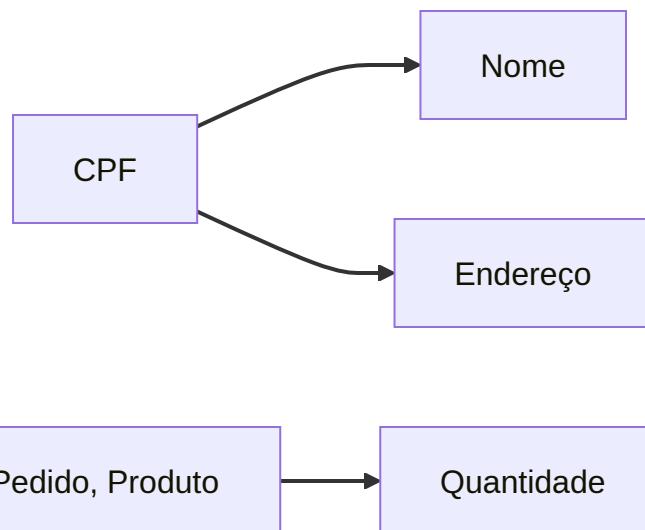
-- Projeção
SELECT nome, cargo FROM Funcionario;

-- Junção
SELECT f.nome, d.nome AS departamento
FROM Funcionario f
JOIN Departamento d ON f.depto_id = d.id;

```

Normalização no Modelo Relacional

1. Dependências Funcionais



2. Formas Normais

1. **1FN:** Valores atômicos
2. **2FN:** Dependência total da chave
3. **3FN:** Sem dependências transitivas
4. **BCNF:** Todas as dependências por chave

Vantagens e Limitações

Vantagens

- Simplicidade conceitual
- Independência de dados
- Flexibilidade
- Integridade garantida
- Base matemática sólida

Limitações

- Tipos de dados complexos
- Relacionamentos hierárquicos
- Performance em alguns casos
- Modelagem OO direta

Implementação Prática

1. Estruturas de Dados

```
CREATE TABLE Cliente (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    data_cadastro DATE DEFAULT CURRENT_DATE,
    status CHAR(1) CHECK (status IN ('A', 'I'))
);
```

2. Restrições

```
ALTER TABLE Pedido
ADD CONSTRAINT fk_cliente
FOREIGN KEY (cliente_id)
REFERENCES Cliente(id)
ON DELETE RESTRICT
ON UPDATE CASCADE;
```

Melhores Práticas

1. Design

- Normalização adequada
- Chaves bem definidas
- Integridade referencial
- Tipos de dados apropriados

2. Nomenclatura

- Padrões consistentes
- Nomes descritivos
- Prefixos/sufixos quando necessário
- Documentação clara

3. Performance

- Índices apropriados
- Constraints adequadas
- Tipos de dados otimizados
- Relacionamentos eficientes

Ferramentas de Modelagem

1. CASE Tools

- MySQL Workbench
- Oracle SQL Developer
- pgModeler
- ERwin

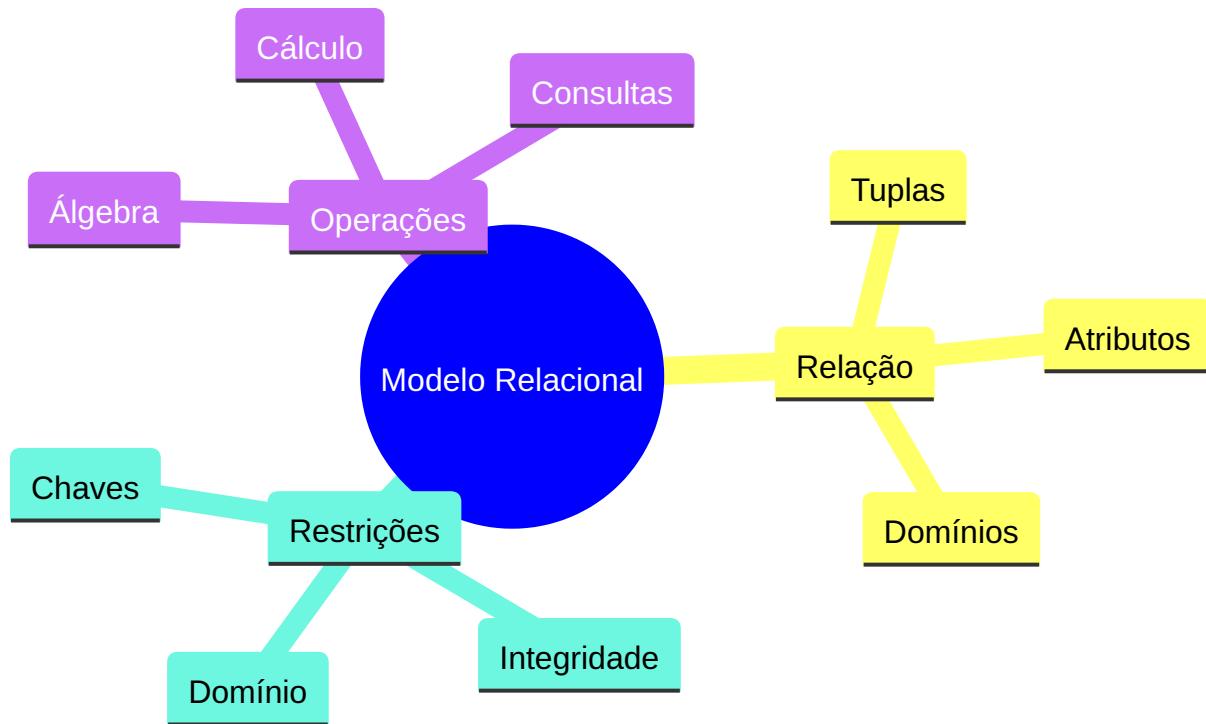
2. Visualização

- DrawSQL
- dbdiagram.io
- QuickDBD
- Lucidchart

Conceitos do Modelo Relacional

O modelo relacional é fundamentado em conceitos matemáticos sólidos que garantem sua consistência e confiabilidade.

Definições Fundamentais



Terminologia

Termo Relacional	Termo Informal	Descrição
Relação	Tabela	Conjunto de tuplas
Tupla	Linha/Registro	Conjunto de valores relacionados
Atributo	Coluna/Campo	Característica da relação
Domínio	Tipo de Dados	Conjunto de valores possíveis
Esquema	Estrutura	Definição da relação
Instância	Dados	Conteúdo atual da relação

Propriedades Essenciais

1. Valores Atômicos
2. Sem Ordem nas Tuplas
3. Sem Duplicatas
4. Sem Ordem nos Atributos
5. Valores Únicos nas Colunas

Exemplos Práticos

Esquema de Relação

```
FUNCIONARIO (ID, Nome, Cargo, Salario, Depto_ID)
```

Instância de Relação

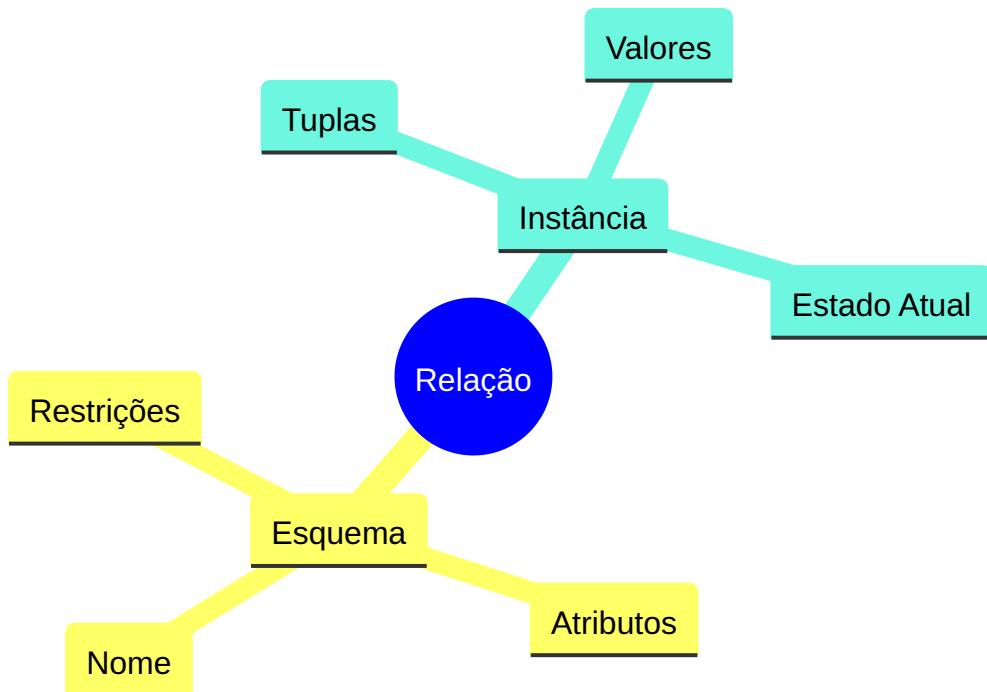
ID	Nome	Cargo	Salario	Depto_ID
1	Ana	Analista	5000	10
2	Carlos	Gerente	8000	20
3	Maria	Desenvolvedora	6000	10

Relações e Tuplas

Conceitos Fundamentais

Relação

Uma relação é uma estrutura matemática que representa uma tabela no modelo relacional.



Propriedades das Relações

1. Sem Ordem nas Tuplas

- A ordem das linhas é irrelevante
- $\{(1,A), (2,B)\} = \{(2,B), (1,A)\}$

2. Sem Duplicatas

- Cada tupla é única
- Identificada pela chave primária

3. Valores Atômicos

- Cada célula contém valor indivisível
- Não permite arrays ou estruturas

Tuplas

Definição Formal

Uma tupla é um conjunto ordenado de valores: $t = (v_1, v_2, \dots, v_n)$

- v_1 pertence ao domínio do primeiro atributo
- v_2 pertence ao domínio do segundo atributo
- v_n pertence ao domínio do n-ésimo atributo

Exemplo Prático

FUNCIONARIO		
int	id	PK
string	nome	
float	salario	
string	depto	

Instância da Relação:

ID	NOME	SALARIO	DEPTO
1	Ana	5000.00	TI
2	Carlos	6000.00	RH
3	Maria	5500.00	TI

Operações com Tuplas

1. Inserção

- Adiciona nova tupla

- Deve respeitar restrições

2. Remoção

- Elimina tupla existente
- Considera integridade referencial

3. Atualização

- Modifica valores
- Mantém consistência

Notação Matemática

Definição de Relação

$$R \subseteq D_1 \times D_2 \times \dots \times D_n$$

- R é a relação
- D_i são os domínios dos atributos

Operações de Conjunto

- União: $R \cup S$
- Interseção: $R \cap S$
- Diferença: $R - S$

Exemplos Detalhados

Sistema de Biblioteca

LIVRO		
string	isbn	PK
string	titulo	
string	autor	
int	ano	

Instâncias:

ISBN	TITULO	AUTOR	ANO
123456789	Banco de Dados	Silva	2020
987654321	SQL Avançado	Pereira	2021

Sistema de Vendas

PEDIDO		
int	numero	PK
date	data	
float	total	
string	status	

Considerações Práticas

1. Integridade dos Dados

- Validação de entrada
- Consistência das tuplas
- Restrições de domínio

2. Performance

- Indexação adequada
- Otimização de consultas

- Gerenciamento de espaço

3. Manutenibilidade

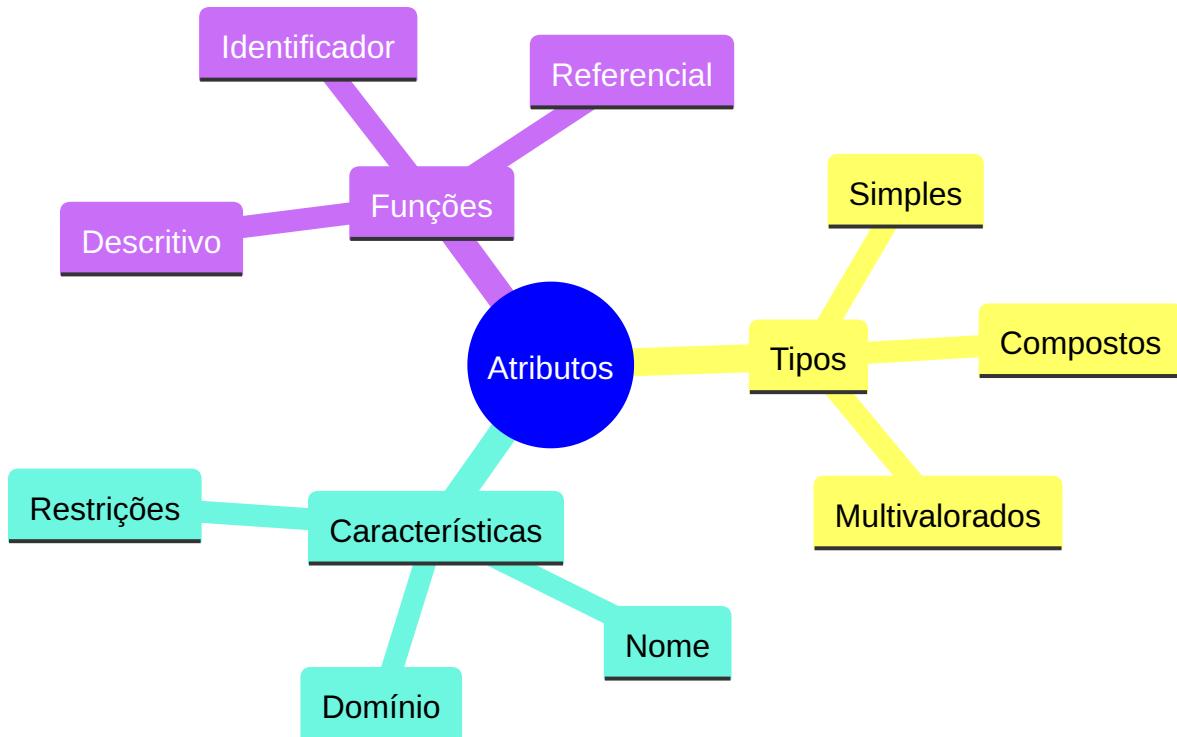
- Documentação clara
- Normalização apropriada
- Padrões de nomenclatura

Atributos e Domínios

Atributos

Definição

Um atributo é uma característica ou propriedade que descreve uma entidade.



Classificação dos Atributos

1. Quanto à Estrutura

- **Simples (Atômicos)**
 - Indivisíveis
 - Exemplo: CPF, idade
- **Compostos**
 - Divisíveis em partes
 - Exemplo: endereço (rua, número, cidade)

- **Multivalorados**
 - Múltiplos valores
 - Exemplo: telefones, emails

2. Quanto à Função

- **Identificadores (Chaves)**
 - Chave Primária (PK)
 - Chave Estrangeira (FK)
 - Chave Única (UK)
- **Descriptivos**
 - Características
 - Propriedades

Domínios

Definição

Um domínio é o conjunto de valores possíveis para um atributo.

Tipos de Domínios

DOMINIO	
string	nome
string	tipo
string	restricoes
string	padrao

1. Domínios Básicos

- Números inteiros
- Números reais

- Texto
- Data/hora
- Booleano

2. Domínios Personalizados

- Enumerações
- Intervalos
- Padrões

Exemplo Prático

```

CREATE DOMAIN Email AS VARCHAR(100)
    CHECK (VALUE ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\$');

CREATE DOMAIN Idade AS INTEGER
    CHECK (VALUE >= 0 AND VALUE <= 150);

CREATE DOMAIN StatusPedido AS VARCHAR(20)
    CHECK (VALUE IN ('Pendente', 'Aprovado', 'Cancelado'));

```

Restrições de Domínio

Tipos de Restrições

1. Tipo de Dado

- INTEGER
- VARCHAR
- DATE
- DECIMAL

2. Intervalo de Valores

- CHECK constraints
- Mínimo/Máximo
- Enumerações

3. Formato

- Expressões regulares
- Padrões específicos
- Máscaras

Exemplos de Implementação

FUNCIONARIO			
string	matricula	PK	Formato: F-\d{5}
string	nome		2 a 100 caracteres
int	idade		18 a 70 anos
string	email		Formato válido
decimal	salario		> 0

Boas Práticas

1. Definição de Atributos

- Nomes significativos
- Tipos apropriados
- Restrições adequadas

2. Gerenciamento de Domínios

- Reutilização
- Consistência
- Documentação

3. Validação de Dados

- Regras de negócio
- Integridade
- Performance

Exemplos Detalhados

Sistema Acadêmico

ALUNO			
string	matricula	PK	A-\d{8}
string	nome		NOT NULL
string	email	UK	formato@dominio
float	cr		0.0 a 10.0
int	periodo		1 a 10

Sistema Financeiro

CONTA			
string	numero	PK	\d{5}-\d
decimal	saldo		>= 0
string	tipo		C/P/E
date	abertura		NOT NULL
boolean	ativa		true/false

Considerações Importantes

1. Integridade

- Validações consistentes
- Regras de negócio
- Consistência dos dados

2. Performance

- Tipos eficientes
- Índices apropriados
- Otimização

3. Manutenibilidade

- Documentação clara
- Padrões consistentes
- Evolução controlada

Restrições de Chave

Tipos de Chaves

1. Chave Primária (Primary Key - PK)

ENTIDADE			
string	id	PK	Identificador único
string	nome		
string	descricao		

Características

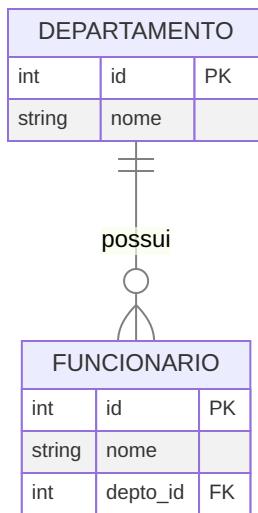
- Identifica unicamente cada tupla
- Não permite valores nulos
- Imutável
- Pode ser simples ou composta

Exemplos

```
CREATE TABLE Produto (
    codigo SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    preco DECIMAL(10,2)
);

-- Chave composta
CREATE TABLE ItemPedido (
    pedido_id INTEGER,
    produto_id INTEGER,
    quantidade INTEGER,
    PRIMARY KEY (pedido_id, produto_id)
);
```

2. Chave Estrangeira (Foreign Key - FK)



Características

- Referencia chave primária de outra tabela
- Mantém integridade referencial
- Pode ser nula (relacionamento opcional)
- Suporta ações referenciais (CASCADE, SET NULL, etc.)

Exemplos

```
CREATE TABLE Funcionario (
    id INTEGER PRIMARY KEY,
    nome VARCHAR(100),
    depto_id INTEGER,
    FOREIGN KEY (depto_id)
        REFERENCES Departamento(id)
        ON DELETE SET NULL
        ON UPDATE CASCADE
);
```

3. Chave Única (Unique Key - UK)

USUARIO			
int	id	PK	
string	email	UK	Único por usuário
string	username	UK	Único no sistema
string	senha		

Características

- Garante unicidade dos valores
- Permite valores nulos (diferente da PK)
- Múltiplas por tabela
- Pode ser composta

Exemplos

```
CREATE TABLE Usuario (
    id SERIAL PRIMARY KEY,
    email VARCHAR(100) UNIQUE,
    username VARCHAR(50) UNIQUE,
    senha VARCHAR(255)
);
```

Implementação em ORMs

1. JPA/Hibernate (Java)

```
@Entity
public class Cliente {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true)
    private String email;
```

```
    @ManyToOne  
    @JoinColumn(name = "categoria_id")  
    private Categoria categoria;  
}
```

2. Django ORM (Python)

```
class Produto(models.Model):  
    codigo = models.CharField(max_length=20, primary_key=True)  
    sku = models.CharField(max_length=50, unique=True)  
    categoria = models.ForeignKey(  
        'Categoria',  
        on_delete=models.CASCADE  
)
```

Boas Práticas

1. Escolha de Chaves Primárias

- Use valores naturais quando apropriado
- Considere surrogate keys para flexibilidade
- Evite chaves compostas complexas
- Mantenha a imutabilidade

2. Gestão de Chaves Estrangeiras

- Defina ações referenciais apropriadas
- Considere o impacto na integridade
- Use índices para performance
- Documente relacionamentos

3. Unicidade

- Identifique campos que exigem unicidade
- Implemente validações em múltiplas camadas
- Considere unicidade combinada
- Trate conflitos adequadamente

Padrões Comuns

1. Chaves Naturais vs Surrogate

PRODUTO_NATURAL			
string	codigo	PK	Chave natural
string	nome		

PRODUTO_SURROGATE			
int	id	PK	Chave surrogate
string	codigo	UK	Chave natural
string	nome		

2. Chaves Compostas

MATRICULA		
string	aluno_id	PK,FK
string	disciplina_id	PK,FK
string	semestre	PK
float	nota	

Considerações de Performance

1. Indexação

- Índices automáticos em PKs
- Índices opcionais em FKs
- Índices únicos para UKs
- Impacto em inserções/atualizações

2. Joins

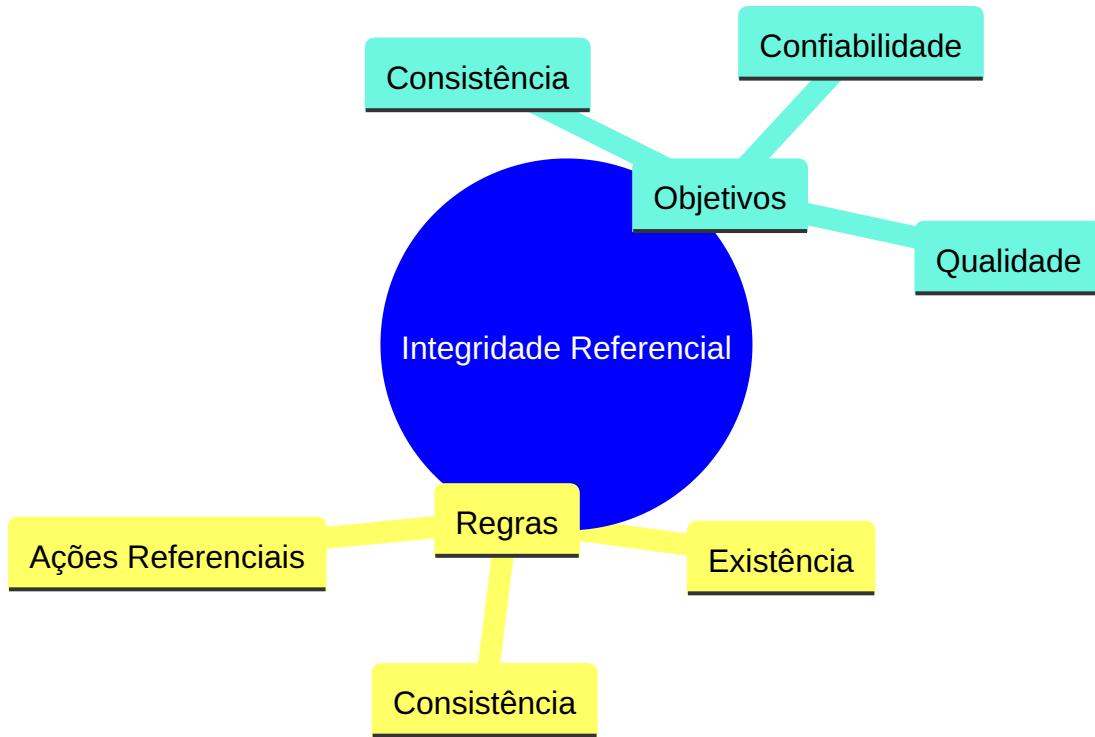
- Otimização de consultas
- Cardinalidade das relações
- Estratégias de indexação
- Planos de execução

Integridade Referencial

Conceitos Fundamentais

Definição

A integridade referencial garante que relacionamentos entre tabelas permaneçam consistentes.



Regras de Integridade

1. Regra de Existência

- FK deve corresponder a PK válida
- Ou ser NULL (se permitido)

2. Regra de Modificação

- Atualizações devem manter consistência

- Deleções devem ser controladas

3. Ações Referenciais

ON DELETE

- CASCADE: Deleta registros relacionados
- SET NULL: Define FK como NULL
- RESTRICT: Impede deleção
- NO ACTION: Comportamento padrão

ON UPDATE

- CASCADE: Atualiza registros relacionados
- SET NULL: Define FK como NULL
- RESTRICT: Impede atualização
- NO ACTION: Comportamento padrão

Implementação

1. SQL DDL

```
CREATE TABLE Pedido (
    id SERIAL PRIMARY KEY,
    cliente_id INTEGER,
    data DATE,
    FOREIGN KEY (cliente_id)
        REFERENCES Cliente(id)
        ON DELETE RESTRICT
        ON UPDATE CASCADE
);
```

2. Constraints Deferidas

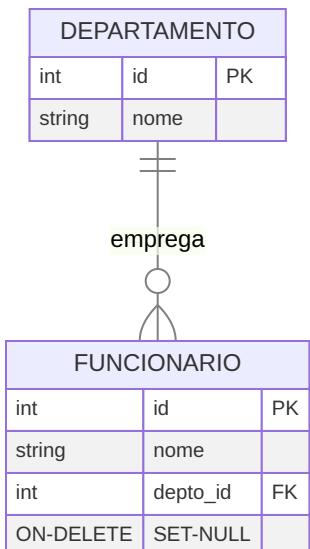
```

CREATE TABLE ItemPedido (
    pedido_id INTEGER,
    produto_id INTEGER,
    quantidade INTEGER,
    FOREIGN KEY (pedido_id)
        REFERENCES Pedido(id)
        DEFERRABLE INITIALLY DEFERRED
);

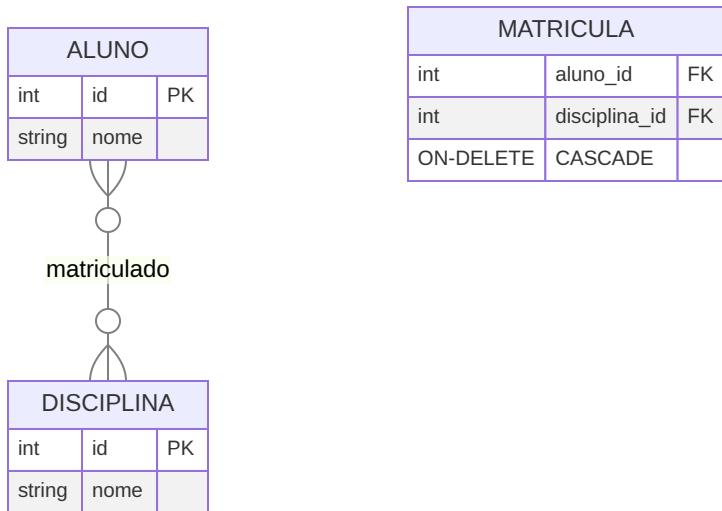
```

Cenários Comuns

1. Relacionamentos Pai-Filho



2. Relacionamentos Muitos-para-Muitos



Boas Práticas

1. Design de Esquema

- Planeje ações referenciais
- Documente decisões
- Considere impacto em cascata

2. Performance

- Use índices apropriados
- Monitore operações em cascata
- Otimize consultas relacionadas

3. Manutenção

- Audite violações
- Mantenha consistência
- Atualize documentação

Tratamento de Erros

1. Violações de Integridade

```
-- Tratamento em transação
BEGIN;
    DELETE FROM Departamento WHERE id = 1;
EXCEPTION WHEN foreign_keyViolation THEN
    -- Tratamento do erro
ROLLBACK;
```

2. Validações Preventivas

```
-- Verificar antes de deletar
SELECT COUNT(*) FROM Funcionario
WHERE depto_id = 1;
```

Monitoramento

1. Logs de Violação

- Registre tentativas falhas
- Analise padrões
- Identifique problemas

2. Métricas

- Taxa de violações
- Performance de operações
- Impacto em cascata

Considerações Avançadas

1. Transações Distribuídas

- Consistência entre sistemas
- Recuperação de falhas
- Sincronização

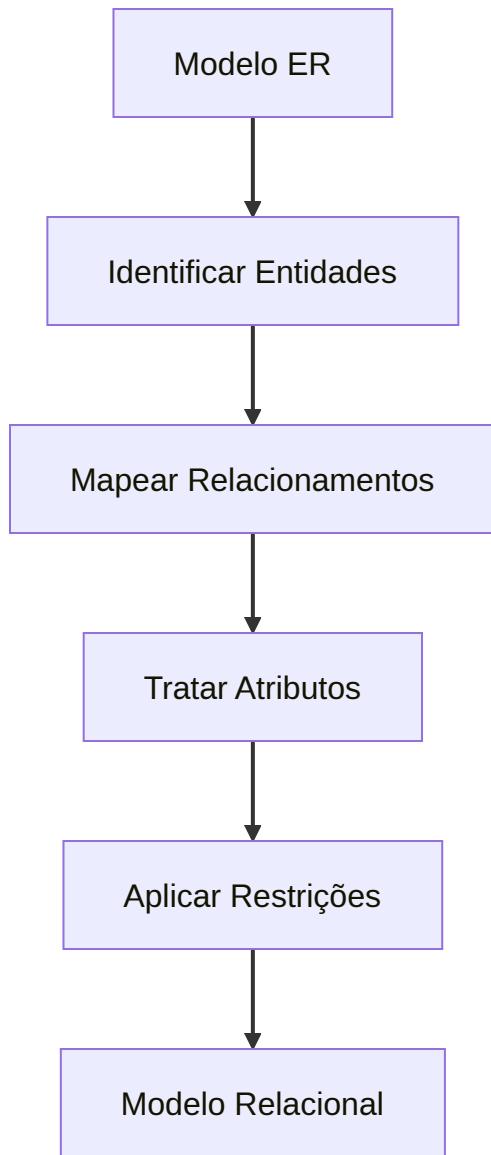
2. Replicação

- Propagação de mudanças
- Consistência eventual
- Resolução de conflitos

Transformação ER para Relacional

A transformação de um modelo ER para o modelo relacional é um processo sistemático que segue regras bem definidas.

Processo de Transformação



Regras de Mapeamento

1. Entidades para Tabelas

CLIENTE		
int	id	PK
string	nome	
string	email	

↓ Transformação ↓

```
CREATE TABLE Cliente (
    id INT PRIMARY KEY,
    nome VARCHAR(100),
    email VARCHAR(100)
);
```

2. Relacionamentos

Cardinalidade 1:1

- Chave estrangeira em qualquer lado
- Preferência para o lado opcional

Cardinalidade 1:N

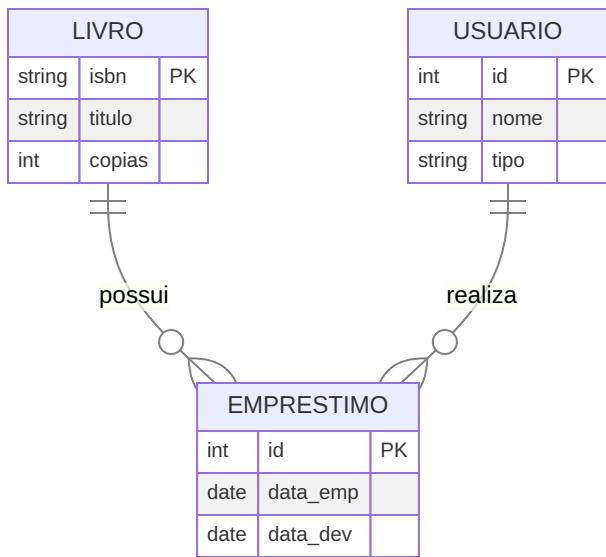
- Chave estrangeira no lado N
- Restrição de integridade referencial

Cardinalidade N:M

- Nova tabela de relacionamento
- Chaves estrangeiras para ambas entidades

Exemplos Detalhados

Sistema de Biblioteca



↓ Transformação ↓

```

CREATE TABLE Livro (
    isbn VARCHAR(13) PRIMARY KEY,
    titulo VARCHAR(200) NOT NULL,
    copias INT DEFAULT 1
);

CREATE TABLE Usuario (
    id INT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    tipo CHAR(1) CHECK (tipo IN ('A', 'P'))
);

CREATE TABLE Emprestimo (
    id INT PRIMARY KEY,
    livro_isbn VARCHAR(13),
    usuario_id INT,
    data_emp DATE NOT NULL,
    data_dev DATE,
    FOREIGN KEY (livro_isbn) REFERENCES Livro(isbn),
    FOREIGN KEY (usuario_id) REFERENCES Usuario(id)
);

```

Checklist de Transformação

1. Preparação

- [] Validar modelo ER
- [] Identificar todas entidades
- [] Listar relacionamentos
- [] Catalogar atributos especiais

2. Execução

- [] Criar tabelas base
- [] Estabelecer chaves primárias
- [] Mapear relacionamentos
- [] Adicionar chaves estrangeiras

3. Validação

- [] Verificar integridade referencial
- [] Confirmar cardinalidades
- [] Testar restrições
- [] Validar normalização

Mapeamento de Entidades ER para Relacional

Regras Fundamentais

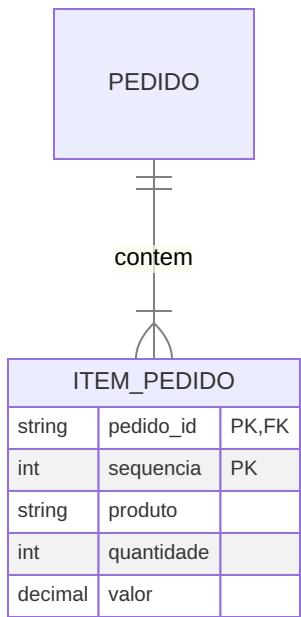
1. Entidades Fortes

CLIENTE		
string	id	PK
string	nome	
string	email	
date	data_cadastro	

↓ Transformação ↓

```
CREATE TABLE Cliente (
    id VARCHAR(50) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    data_cadastro DATE NOT NULL
);
```

2. Entidades Fracas



↓ Transformação ↓

```

CREATE TABLE Item_Pedido (
    pedido_id VARCHAR(50),
    sequencia INTEGER,
    produto VARCHAR(100) NOT NULL,
    quantidade INTEGER NOT NULL,
    valor DECIMAL(10,2) NOT NULL,
    PRIMARY KEY (pedido_id, sequencia),
    FOREIGN KEY (pedido_id) REFERENCES Pedido(id)
);
  
```

Mapeamento de Atributos

1. Atributos Simples

- Mapeamento direto para colunas
- Tipo de dados apropriado
- Restrições básicas (NOT NULL, etc.)

2. Atributos Compostos

ENDERECO		
string	id	PK
string	rua	
string	numero	
string	complemento	
string	bairro	
string	cidade	
string	estado	
string	cep	

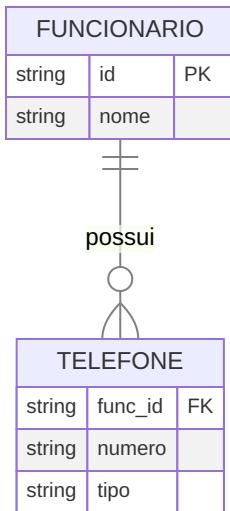
Opção 1: Decomposição

```
CREATE TABLE Endereco (
    id VARCHAR(50) PRIMARY KEY,
    rua VARCHAR(100),
    numero VARCHAR(10),
    complemento VARCHAR(50),
    bairro VARCHAR(50),
    cidade VARCHAR(50),
    estado CHAR(2),
    cep VARCHAR(8)
);
```

Opção 2: Agregação

```
CREATE TABLE Endereco (
    id VARCHAR(50) PRIMARY KEY,
    endereco_completo VARCHAR(300)
);
```

3. Atributos Multivalorados



```

CREATE TABLE Telefone (
    func_id VARCHAR(50),
    numero VARCHAR(20),
    tipo VARCHAR(20),
    PRIMARY KEY (func_id, numero),
    FOREIGN KEY (func_id) REFERENCES Funcionario(id)
);

```

Estratégias de Identificação

1. Chaves Naturais

```

CREATE TABLE Produto (
    codigo_barras VARCHAR(13) PRIMARY KEY,
    descricao VARCHAR(200),
    preco DECIMAL(10,2)
);

```

2. Chaves Surrogate

```

CREATE TABLE Produto (
    id SERIAL PRIMARY KEY,
    codigo_barras VARCHAR(13) UNIQUE,
    descricao VARCHAR(200),

```

```
    preco DECIMAL(10, 2)
);
```

Restrições e Integridade

1. Domínios

```
CREATE DOMAIN Email AS VARCHAR(100)
  CHECK (VALUE ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$');
```



```
CREATE TABLE Usuario (
  id SERIAL PRIMARY KEY,
  email Email NOT NULL UNIQUE
);
```

2. Checks

```
CREATE TABLE Produto (
  id SERIAL PRIMARY KEY,
  nome VARCHAR(100),
  preco DECIMAL(10, 2),
  estoque INTEGER,
  CONSTRAINT check_preco_positivo CHECK (preco > 0),
  CONSTRAINT check_estoque_positivo CHECK (estoque >= 0)
);
```

Padrões de Implementação

1. Auditoria

```
CREATE TABLE Cliente (
  id SERIAL PRIMARY KEY,
  nome VARCHAR(100),
  criado_em TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  atualizado_em TIMESTAMP,
```

```
    criado_por VARCHAR(50),
    atualizado_por VARCHAR(50)
);
```

2. Soft Delete

```
CREATE TABLE Produto (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100),
    ativo BOOLEAN DEFAULT true,
    deletado_em TIMESTAMP,
    deletado_por VARCHAR(50)
);
```

Considerações de Performance

1. Índices

```
CREATE TABLE Pedido (
    id SERIAL PRIMARY KEY,
    cliente_id INTEGER,
    data DATE,
    status VARCHAR(20)
);

CREATE INDEX idx_pedido_cliente ON Pedido(cliente_id);
CREATE INDEX idx_pedido_data ON Pedido(data);
CREATE INDEX idx_pedido_status ON Pedido(status);
```

2. Particionamento

```
CREATE TABLE Vendas (
    id SERIAL,
    data DATE,
    valor DECIMAL(10,2)
) PARTITION BY RANGE (data);
```

```
CREATE TABLE vendas_2023 PARTITION OF Vendas
    FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');
```

Exemplos Práticos

1. Sistema de E-commerce

```
CREATE TABLE Produto (
    id SERIAL PRIMARY KEY,
    sku VARCHAR(50) UNIQUE,
    nome VARCHAR(100),
    descricao TEXT,
    preco DECIMAL(10,2),
    estoque INTEGER,
    categoria_id INTEGER REFERENCES Categoria(id)
);

CREATE TABLE Pedido (
    id SERIAL PRIMARY KEY,
    cliente_id INTEGER REFERENCES Cliente(id),
    data TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status VARCHAR(20),
    valor_total DECIMAL(10,2)
);

CREATE TABLE Item_Pedido (
    pedido_id INTEGER,
    produto_id INTEGER,
    quantidade INTEGER,
    preco_unitario DECIMAL(10,2),
    PRIMARY KEY (pedido_id, produto_id),
    FOREIGN KEY (pedido_id) REFERENCES Pedido(id),
    FOREIGN KEY (produto_id) REFERENCES Produto(id)
);
```

2. Sistema Acadêmico

```
CREATE TABLE Aluno (
    matricula VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100),
    data_nascimento DATE,
    curso_id INTEGER REFERENCES Curso(id)
);

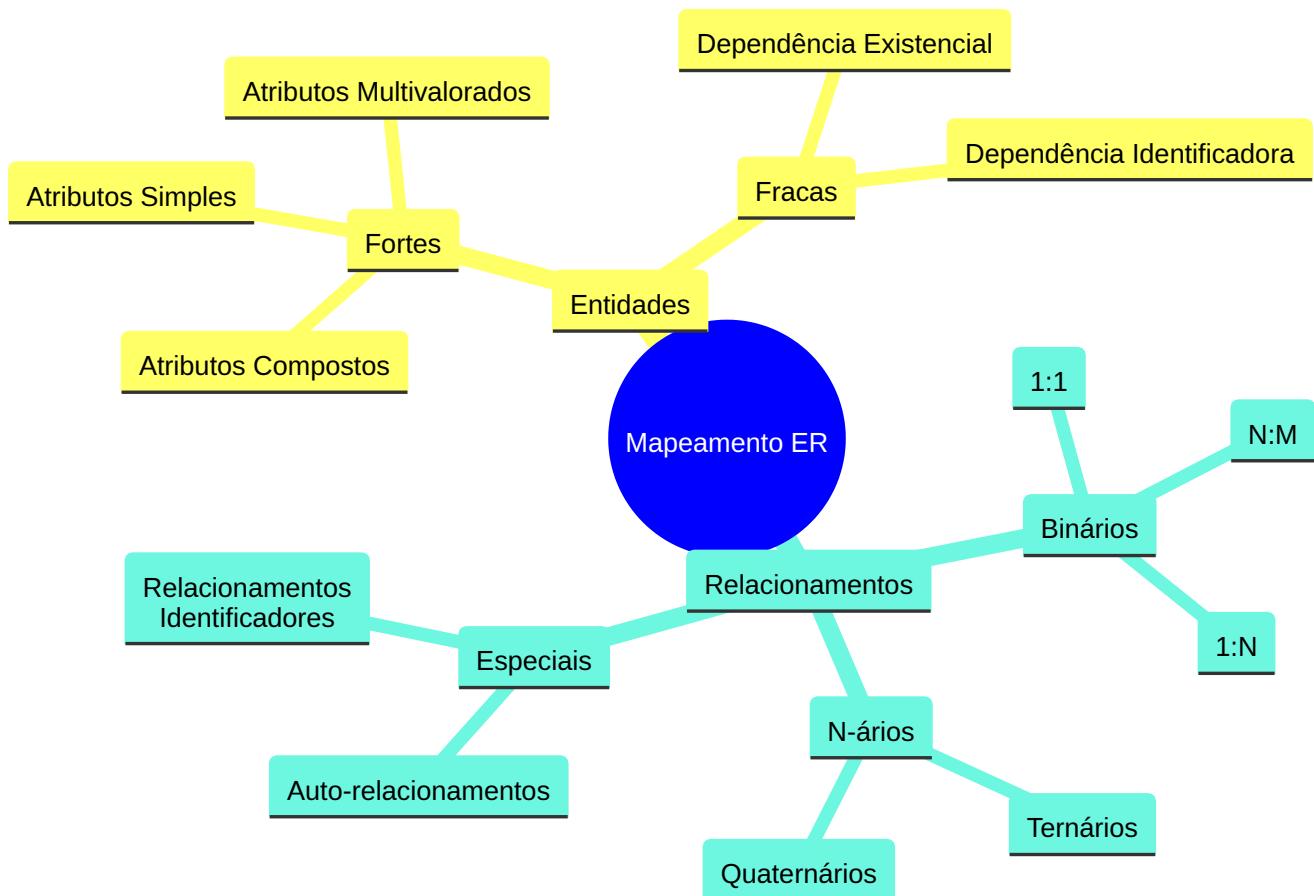
CREATE TABLE Disciplina (
    codigo VARCHAR(10) PRIMARY KEY,
    nome VARCHAR(100),
    carga_horaria INTEGER,
    departamento_id INTEGER REFERENCES Departamento(id)
);

CREATE TABLE Matricula (
    aluno_id VARCHAR(20),
    disciplina_id VARCHAR(10),
    semestre VARCHAR(6),
    nota DECIMAL(4, 2),
    frequencia INTEGER,
    PRIMARY KEY (aluno_id, disciplina_id, semestre),
    FOREIGN KEY (aluno_id) REFERENCES Aluno(matricula),
    FOREIGN KEY (disciplina_id) REFERENCES Disciplina(codigo)
);
```

Mapeamento de Relacionamentos ER para Relacional

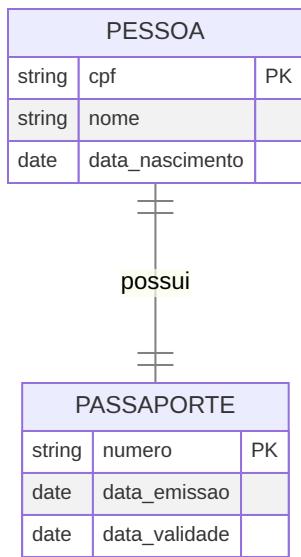
Fundamentos do Mapeamento

Conceitos Básicos



Mapeamento por Cardinalidade

1. Relacionamentos 1:1



Implementação

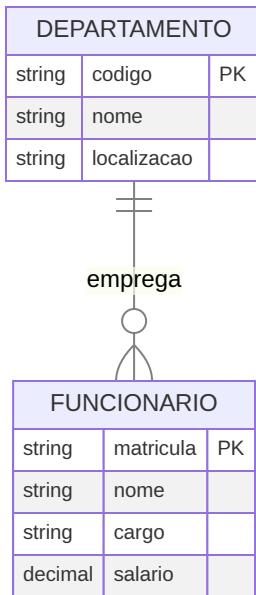
```

CREATE TABLE Pessoa (
    cpf VARCHAR(11) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    data_nascimento DATE,
    passaporte_numero VARCHAR(20) UNIQUE,
    FOREIGN KEY (passaporte_numero) REFERENCES Passaporte(numero)
);

CREATE TABLE Passaporte (
    numero VARCHAR(20) PRIMARY KEY,
    data_emissao DATE NOT NULL,
    data_validade DATE NOT NULL
);

```

2. Relacionamentos 1:N



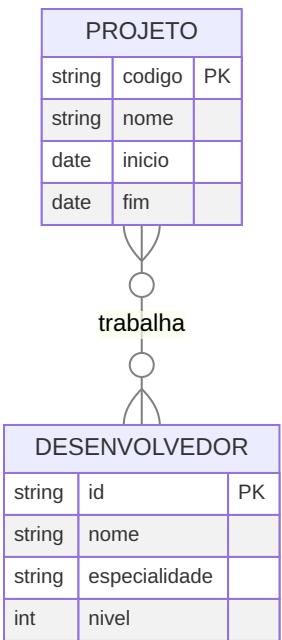
Implementação

```

CREATE TABLE Departamento (
    codigo VARCHAR(10) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    localizacao VARCHAR(100)
);

CREATE TABLE Funcionario (
    matricula VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    cargo VARCHAR(50),
    salario DECIMAL(10, 2),
    departamento_codigo VARCHAR(10) NOT NULL,
    FOREIGN KEY (departamento_codigo) REFERENCES
Departamento(codigo)
);
  
```

3. Relacionamentos N:M



Implementação

```

CREATE TABLE Projeto (
    codigo VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    inicio DATE,
    fim DATE
);

```

```

CREATE TABLE Desenvolvedor (
    id VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    especialidade VARCHAR(50),
    nivel INTEGER
);

```

```

CREATE TABLE Projeto_Desenvolvedor (
    projeto_codigo VARCHAR(20),
    desenvolvedor_id VARCHAR(20),
    data_alocacao DATE NOT NULL,
    horas_semanais INTEGER,
    papel VARCHAR(50),
    ...
);

```

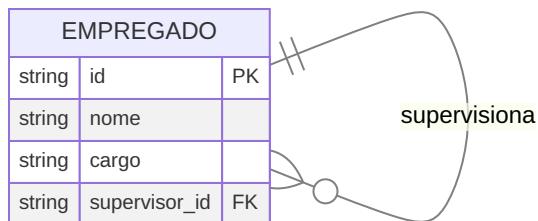
```

    PRIMARY KEY (projeto_codigo, desenvolvedor_id),
    FOREIGN KEY (projeto_codigo) REFERENCES Projeto(codigo),
    FOREIGN KEY (desenvolvedor_id) REFERENCES Desenvolvedor(id)
);

```

Casos Especiais

1. Auto-relacionamentos



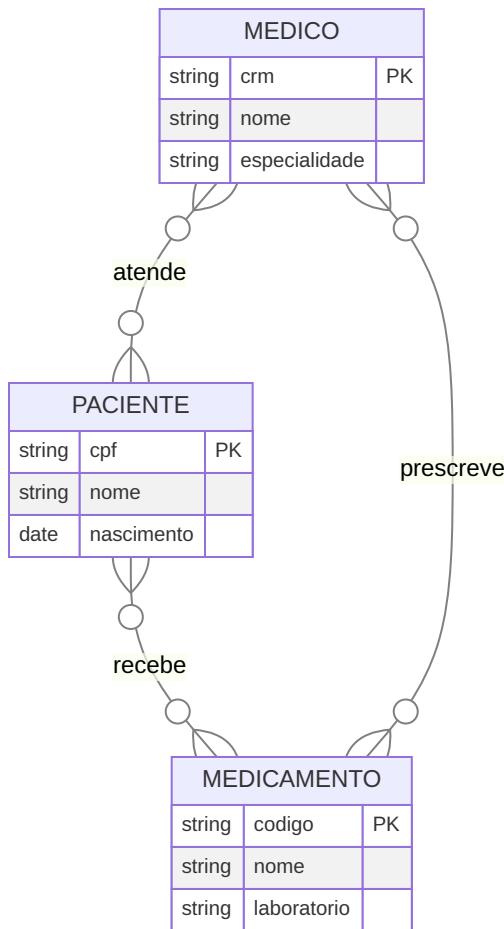
Implementação

```

CREATE TABLE Empregado (
    id VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    cargo VARCHAR(50),
    supervisor_id VARCHAR(20),
    FOREIGN KEY (supervisor_id) REFERENCES Empregado(id)
);

```

2. Relacionamentos Ternários



Implementação

```

CREATE TABLE Medico (
    crm VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    especialidade VARCHAR(50)
);

```

```

CREATE TABLE Paciente (
    cpf VARCHAR(11) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    nascimento DATE
);

```

```

CREATE TABLE Medicamento (
    codigo VARCHAR(20) PRIMARY KEY,

```

```

        nome VARCHAR(100) NOT NULL,
        laboratorio VARCHAR(100)
);

CREATE TABLE Prescricao (
    medico_crm VARCHAR(20),
    paciente_cpf VARCHAR(11),
    medicamento_codigo VARCHAR(20),
    data_prescricao DATE NOT NULL,
    dosagem VARCHAR(50),
    duracao_dias INTEGER,
    PRIMARY KEY (medico_crm, paciente_cpf, medicamento_codigo,
data_prescricao),
    FOREIGN KEY (medico_crm) REFERENCES Medico(crm),
    FOREIGN KEY (paciente_cpf) REFERENCES Paciente(cpf),
    FOREIGN KEY (medicamento_codigo) REFERENCES
Medicamento(codigo)
);

```

Otimizações

1. Índices

```

-- Índices para relacionamentos 1:N
CREATE INDEX idx_funcionario_dept ON
Funcionario(departamento_codigo);

-- Índices para relacionamentos N:M
CREATE INDEX idx_proj_dev_proj ON
Projeto_Desenvolvedor(projeto_codigo);
CREATE INDEX idx_proj_dev_dev ON
Projeto_Desenvolvedor(desenvolvedor_id);

```

2. Constraints

```
-- Restrições de integridade
ALTER TABLE Funcionario
ADD CONSTRAINT check_salario CHECK (salario > 0);

-- Restrições de exclusão
ALTER TABLE Funcionario
ADD CONSTRAINT fk_dept
FOREIGN KEY (departamento_codigo)
REFERENCES Departamento(codigo)
ON DELETE RESTRICT
ON UPDATE CASCADE;
```

Boas Práticas

1. Nomenclatura

- Nomes descritivos para tabelas de relacionamento
- Prefixos consistentes para chaves estrangeiras
- Sufixos padronizados para índices e constraints

2. Documentação

- Comentários explicativos nas tabelas
- Documentação das regras de negócio
- Diagramas de relacionamento

3. Performance

- Análise de cardinalidade
- Estratégia de indexação
- Monitoramento de consultas

Exemplos Práticos

Sistema de Biblioteca

```
CREATE TABLE Livro (
    isbn VARCHAR(13) PRIMARY KEY,
    titulo VARCHAR(200) NOT NULL,
    ano INTEGER,
    copias_disponiveis INTEGER
);

CREATE TABLE Usuario (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE
);

CREATE TABLE Emprestimo (
    livro_isbn VARCHAR(13),
    usuario_id INTEGER,
    data_emprestimo DATE NOT NULL,
    data_devolucao_prevista DATE NOT NULL,
    data_devolucao_real DATE,
    PRIMARY KEY (livro_isbn, usuario_id, data_emprestimo),
    FOREIGN KEY (livro_isbn) REFERENCES Livro(isbn),
    FOREIGN KEY (usuario_id) REFERENCES Usuario(id)
);
```

Sistema de E-commerce

```
CREATE TABLE Pedido (
    numero VARCHAR(20) PRIMARY KEY,
    cliente_id INTEGER REFERENCES Cliente(id),
    data_pedido TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status VARCHAR(20)
);
```

```
CREATE TABLE Produto (
    codigo VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    preco DECIMAL(10,2)
);

CREATE TABLE Item_Pedido (
    pedido_numero VARCHAR(20),
    produto_codigo VARCHAR(20),
    quantidade INTEGER NOT NULL,
    preco_unitario DECIMAL(10,2) NOT NULL,
    PRIMARY KEY (pedido_numero, produto_codigo),
    FOREIGN KEY (pedido_numero) REFERENCES Pedido(numero),
    FOREIGN KEY (produto_codigo) REFERENCES Produto(codigo)
);
```

Mapeamento de Atributos ER para Relacional

Tipos de Atributos

1. Atributos Simples

PRODUTO		
string	codigo	PK
string	nome	
decimal	preco	
int	quantidade	

Implementação

```
CREATE TABLE Produto (
    codigo VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    preco DECIMAL(10,2) NOT NULL,
    quantidade INTEGER DEFAULT 0
);
```

2. Atributos Compostos

FUNCIONARIO		
string	id	PK
string	nome	
string	endereco_rua	
string	endereco_numero	
string	endereco_cidade	
string	endereco_estado	
string	endereco_cep	

Método 1: Decomposição

```

CREATE TABLE Funcionario (
    id VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    endereco_rua VARCHAR(100),
    endereco_numero VARCHAR(10),
    endereco_cidade VARCHAR(50),
    endereco_estado CHAR(2),
    endereco_cep VARCHAR(8)
);

```

Método 2: Nova Entidade

```

CREATE TABLE Funcionario (
    id VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL
);

CREATE TABLE Endereco (
    funcionario_id VARCHAR(20) PRIMARY KEY,
    rua VARCHAR(100),
    numero VARCHAR(10),
    cidade VARCHAR(50),
    estado CHAR(2),
    cep VARCHAR(8),
    FOREIGN KEY (funcionario_id) REFERENCES Funcionario(id)
);

```

3. Atributos Multivalorados

PESSOA		
string	cpf	PK
string	nome	
string[]	telefones	
string[]	emails	

Implementação

```

CREATE TABLE Pessoa (
    cpf VARCHAR(11) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL
);

CREATE TABLE Telefone_Pessoa (
    pessoa_cpf VARCHAR(11),
    telefone VARCHAR(20),
    tipo VARCHAR(20),
    PRIMARY KEY (pessoa_cpf, telefone),
    FOREIGN KEY (pessoa_cpf) REFERENCES Pessoa(cpf)
);

CREATE TABLE Email_Pessoa (
    pessoa_cpf VARCHAR(11),
    email VARCHAR(100),
    principal BOOLEAN DEFAULT false,
    PRIMARY KEY (pessoa_cpf, email),
    FOREIGN KEY (pessoa_cpf) REFERENCES Pessoa(cpf)
);

```

4. Atributos Derivados

PEDIDO			
string	numero	PK	
decimal	subtotal		
decimal	desconto		
decimal	total		Derivado

Implementação

```

CREATE TABLE Pedido (
    numero VARCHAR(20) PRIMARY KEY,
    subtotal DECIMAL(10,2) NOT NULL,
    desconto DECIMAL(10,2) DEFAULT 0,
    -- total é calculado: subtotal - desconto
    CHECK (desconto >= 0 AND desconto <= subtotal)

```

```
);

CREATE VIEW Pedido_Com_Total AS
SELECT
    numero,
    subtotal,
    desconto,
    (subtotal - desconto) as total
FROM Pedido;
```

Restrições e Validações

1. Domínios Personalizados

```
CREATE DOMAIN Email AS VARCHAR(100)
    CHECK (VALUE ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$');
```

```
CREATE DOMAIN CPF AS VARCHAR(11)
    CHECK (VALUE ~ '^\\d{11}$');
```

```
CREATE TABLE Cliente (
    cpf CPF PRIMARY KEY,
    email Email,
    nome VARCHAR(100) NOT NULL
);
```

2. Restrições de Valor

```
CREATE TABLE Produto (
    codigo VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    preco DECIMAL(10,2),
    estoque INTEGER,
    CONSTRAINT check_preco_positivo CHECK (preco > 0),
```

```
        CONSTRAINT check_estoque_nao_negativo CHECK (estoque >= 0)
);
```

3. Valores Default

```
CREATE TABLE Usuario (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    ativo BOOLEAN DEFAULT true,
    data_cadastro TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    tentativas_login INTEGER DEFAULT 0
);
```

Padrões de Implementação

1. Atributos de Auditoria

```
CREATE TABLE Entidade (
    id VARCHAR(20) PRIMARY KEY,
    -- outros atributos
    criado_em TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    criado_por VARCHAR(50),
    alterado_em TIMESTAMP,
    alterado_por VARCHAR(50)
);
```

```
CREATE TRIGGER atualiza_auditoria
    BEFORE UPDATE ON Entidade
    FOR EACH ROW
    EXECUTE FUNCTION fn_atualiza_auditoria();
```

2. Atributos Sensíveis

```
CREATE TABLE Usuario (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
```

```
email VARCHAR(100) UNIQUE NOT NULL,  
senha_hash VARCHAR(64) NOT NULL,  
dados_sensiveis JSONB ENCRYPTED  
);
```

3. Atributos JSON

```
CREATE TABLE Configuracao (  
    id SERIAL PRIMARY KEY,  
    chave VARCHAR(50) UNIQUE NOT NULL,  
    valor JSONB,  
    metadata JSONB DEFAULT '{}'::jsonb  
);
```

Otimizações

1. Índices

```
-- Índice para busca por texto  
CREATE INDEX idx_produto_nome ON Produto USING GIN  
(to_tsvector('portuguese', nome));  
  
-- Índice para JSON  
CREATE INDEX idx_config_valor ON Configuracao USING GIN (valor  
jsonb_path_ops);  
  
-- Índice parcial  
CREATE INDEX idx_usuario_ativo ON Usuario(email) WHERE ativo =  
true;
```

2. Compressão

```
CREATE TABLE Historico (  
    id BIGSERIAL PRIMARY KEY,  
    dados TEXT COMPRESSION lz4
```

```
)  
TABLESPACE historico_tablespace;
```

Exemplos Práticos

Sistema de Vendas

```
CREATE TABLE Produto (  
    codigo VARCHAR(20) PRIMARY KEY,  
    nome VARCHAR(100) NOT NULL,  
    descricao TEXT,  
    preco_custo DECIMAL(10,2),  
    preco_venda DECIMAL(10,2),  
    margem_lucro DECIMAL(5,2) GENERATED ALWAYS AS (  
        ((preco_venda - preco_custo) / preco_custo) * 100  
    ) STORED,  
    especificacoes JSONB,  
    CONSTRAINT check_precos CHECK (preco_venda > preco_custo)  
);
```

```
CREATE TABLE Categoria (  
    id SERIAL PRIMARY KEY,  
    nome VARCHAR(50) UNIQUE NOT NULL,  
    atributos_especificos JSONB DEFAULT '{} '::jsonb  
);
```

```
CREATE TABLE Produto_Categoria (  
    produto_codigo VARCHAR(20),  
    categoria_id INTEGER,  
    ordem INTEGER DEFAULT 1,  
    PRIMARY KEY (produto_codigo, categoria_id),  
    FOREIGN KEY (produto_codigo) REFERENCES Produto(codigo),  
    FOREIGN KEY (categoria_id) REFERENCES Categoria(id)  
);
```

Considerações Importantes

1. Performance

- Escolha tipos de dados apropriados
- Defina índices estrategicamente
- Monitore o uso de atributos

2. Manutenibilidade

- Documente decisões de design
- Use nomes descritivos
- Mantenha consistência

3. Segurança

- Proteja dados sensíveis
- Implemente auditoria
- Valide entradas

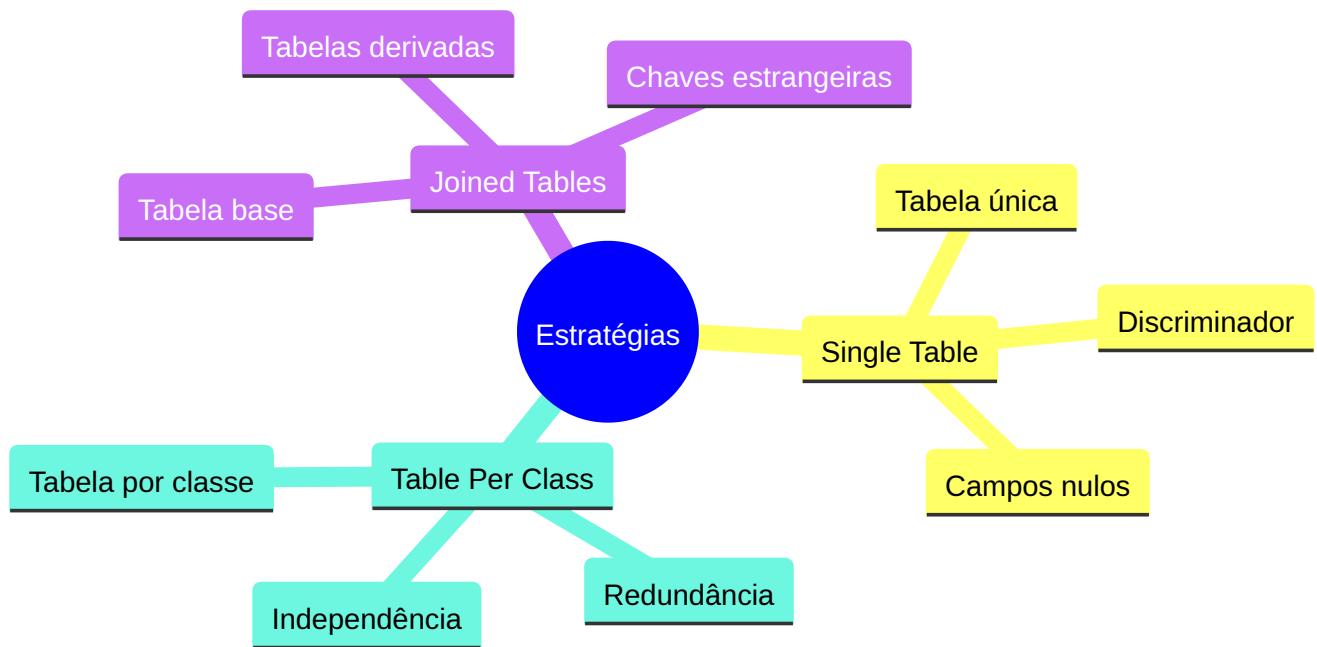
Conclusão

O mapeamento eficiente de atributos é crucial para:

- Integridade dos dados
- Performance do sistema
- Facilidade de manutenção
- Segurança da informação

Mapeamento de Herança ER para Relacional

Visão Geral das Estratégias



Estratégias de Mapeamento

1. Single Table (Tabela Única)

PESSOA			
string	id	PK	
string	nome		
string	tipo		
string	matricula		Aluno
string	siape		Professor
string	departamento		Professor
float	cr		Aluno
string	turma		Aluno
string	titulacao		Professor
string	sala		Professor

Implementação

```
CREATE TABLE Pessoa (
    id VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    tipo VARCHAR(20) NOT NULL,
    -- Atributos de Aluno
    matricula VARCHAR(20),
    cr FLOAT,
    turma VARCHAR(20),
    -- Atributos de Professor
    siape VARCHAR(20),
    departamento VARCHAR(50),
    titulacao VARCHAR(30),
    sala VARCHAR(10),
    -- Restrições
    CONSTRAINT check_tipo CHECK (tipo IN ('ALUNO', 'PROFESSOR')),
    CONSTRAINT check_aluno CHECK (
        (tipo = 'ALUNO' AND matricula IS NOT NULL) OR
        (tipo = 'PROFESSOR' AND matricula IS NULL)
    ),
    CONSTRAINT check_professor CHECK (
        (tipo = 'PROFESSOR' AND siape IS NOT NULL) OR
        (tipo = 'ALUNO' AND siape IS NULL)
    )
);
```

Vantagens

- Consultas simples
- Sem necessidade de junções
- Fácil manutenção

Desvantagens

- Desperdício de espaço

- Muitos campos nulos
- Menor integridade de dados

2. Table Per Class (Tabela por Classe)

ALUNO		
string	id	PK
string	nome	
string	matricula	
float	cr	
string	turma	

PROFESSOR		
string	id	PK
string	nome	
string	siape	
string	departamento	
string	titulacao	
string	sala	

Implementação

```
CREATE TABLE Aluno (
    id VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    matricula VARCHAR(20) UNIQUE NOT NULL,
    cr FLOAT,
    turma VARCHAR(20)
);
```

```
CREATE TABLE Professor (
    id VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    siape VARCHAR(20) UNIQUE NOT NULL,
    departamento VARCHAR(50),
    titulacao VARCHAR(30),
    sala VARCHAR(10)
);
```

Vantagens

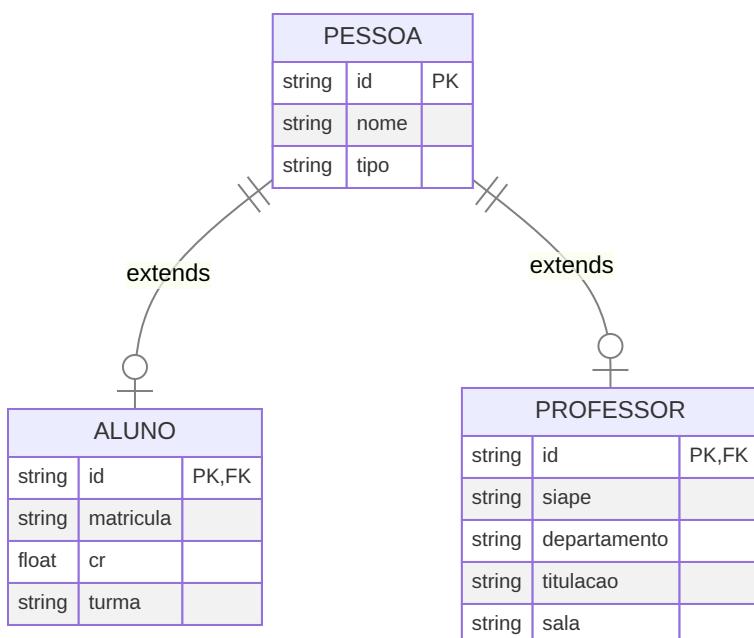
- Modelo mais flexível
- Melhor integridade de dados

- Consultas específicas eficientes

Desvantagens

- Redundância de dados
- Consultas polimórficas complexas
- Maior espaço de armazenamento

3. Joined Tables (Tabelas Unidas)



Implementação

```

CREATE TABLE Pessoa (
    id VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    tipo VARCHAR(20) NOT NULL,
    CONSTRAINT check_tipo CHECK (tipo IN ('ALUNO', 'PROFESSOR'))
);
  
```

```

CREATE TABLE Aluno (
    id VARCHAR(20) PRIMARY KEY,
    matricula VARCHAR(20) UNIQUE NOT NULL,
  
```

```

cr FLOAT,
turma VARCHAR(20),
FOREIGN KEY (id) REFERENCES Pessoa(id)
);

CREATE TABLE Professor (
    id VARCHAR(20) PRIMARY KEY,
    siape VARCHAR(20) UNIQUE NOT NULL,
    departamento VARCHAR(50),
    titulacao VARCHAR(30),
    sala VARCHAR(10),
    FOREIGN KEY (id) REFERENCES Pessoa(id)
);

```

Vantagens

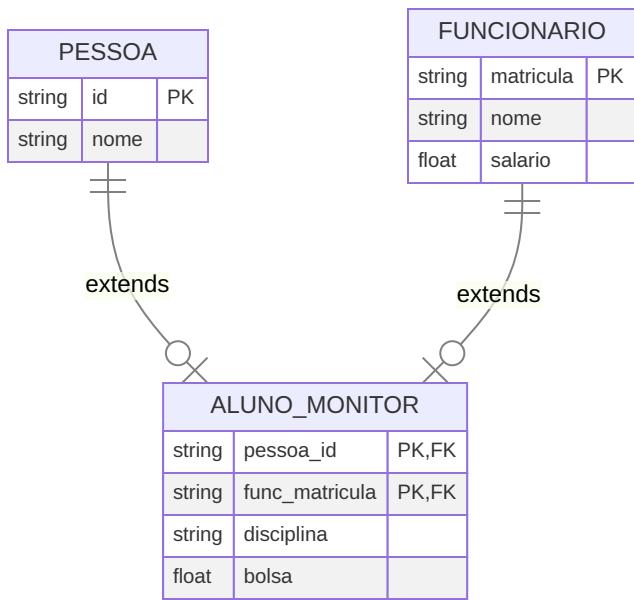
- Normalização completa
- Sem campos nulos
- Integridade referencial

Desvantagens

- Necessidade de junções
- Performance reduzida
- Complexidade de manutenção

Casos Especiais

1. Herança Múltipla



Implementação

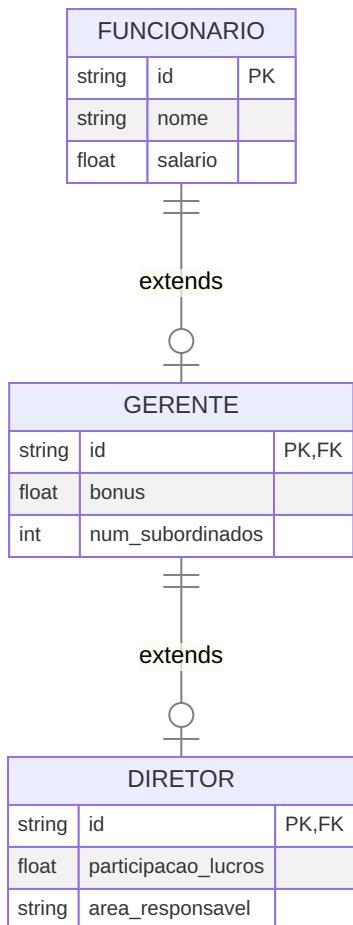
```

CREATE TABLE Pessoa (
    id VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL
);

CREATE TABLE Funcionario (
    matricula VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    salario DECIMAL(10, 2)
);

CREATE TABLE Aluno_Monitor (
    pessoa_id VARCHAR(20),
    func_matricula VARCHAR(20),
    disciplina VARCHAR(50) NOT NULL,
    bolsa DECIMAL(10, 2),
    PRIMARY KEY (pessoa_id, func_matricula),
    FOREIGN KEY (pessoa_id) REFERENCES Pessoa(id),
    FOREIGN KEY (func_matricula) REFERENCES Funcionario(matricula)
);
  
```

2. Herança Hierárquica



Implementação

```
CREATE TABLE Funcionario (
    id VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    salario DECIMAL(10, 2),
    tipo VARCHAR(20) NOT NULL,
    CONSTRAINT check_tipo CHECK (tipo IN ('FUNCIONARIO',
    'GERENTE', 'DIRETOR'))
);
```

```
CREATE TABLE Gerente (
    id VARCHAR(20) PRIMARY KEY,
    bonus DECIMAL(10, 2),
    num_subordinados INTEGER,
```

```

    FOREIGN KEY (id) REFERENCES Funcionario(id)
);

CREATE TABLE Diretor (
    id VARCHAR(20) PRIMARY KEY,
    participacao_lucros DECIMAL(10, 2),
    area_responsavel VARCHAR(50),
    FOREIGN KEY (id) REFERENCES Gerente(id)
);

```

Otimizações

1. Índices

```

-- Índices para junções eficientes
CREATE INDEX idx_pessoa_tipo ON Pessoa(tipo);
CREATE INDEX idx_aluno_matricula ON Aluno(matricula);
CREATE INDEX idx_professor_siape ON Professor(siape);

-- Índices para consultas frequentes
CREATE INDEX idx_funcionario_tipo ON Funcionario(tipo);
CREATE INDEX idx_gerente_subordinados ON
Gerente(num_subordinados);

```

2. Views

```

-- View para consulta unificada de pessoas
CREATE VIEW vw_pessoas AS
    SELECT p.id, p.nome, p.tipo,
           a.matricula, a.cr, a.turma,
           pr.siape, pr.departamento, pr.titulacao
    FROM Pessoa p
    LEFT JOIN Aluno a ON p.id = a.id
    LEFT JOIN Professor pr ON p.id = pr.id;

-- View para hierarquia de funcionários

```

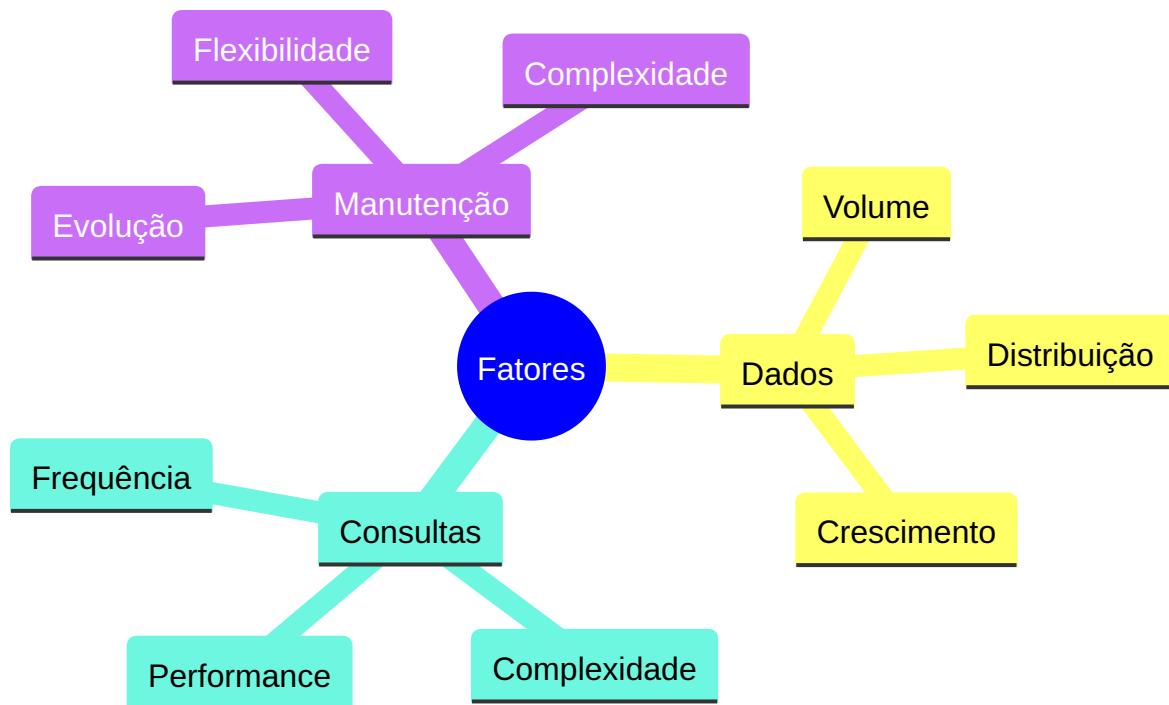
```

CREATE VIEW vw_funcionarios AS
    SELECT f.id, f.nome, f.salario, f.tipo,
           g.bonus, g.num_subordinados,
           d.participacao_lucros, d.area_responsavel
      FROM Funcionario f
     LEFT JOIN Gerente g ON f.id = g.id
    LEFT JOIN Diretor d ON g.id = d.id;

```

Considerações de Design

Escolha da Estratégia



Recomendações

1. Single Table

- Hierarquias simples
- Poucos atributos específicos
- Consultas frequentes polimórficas

2. Table Per Class

- Subclasses muito diferentes
- Consultas específicas frequentes
- Poucos dados compartilhados

3. Joined Tables

- Alta normalização necessária
- Dados compartilhados importantes
- Evolução frequente do esquema

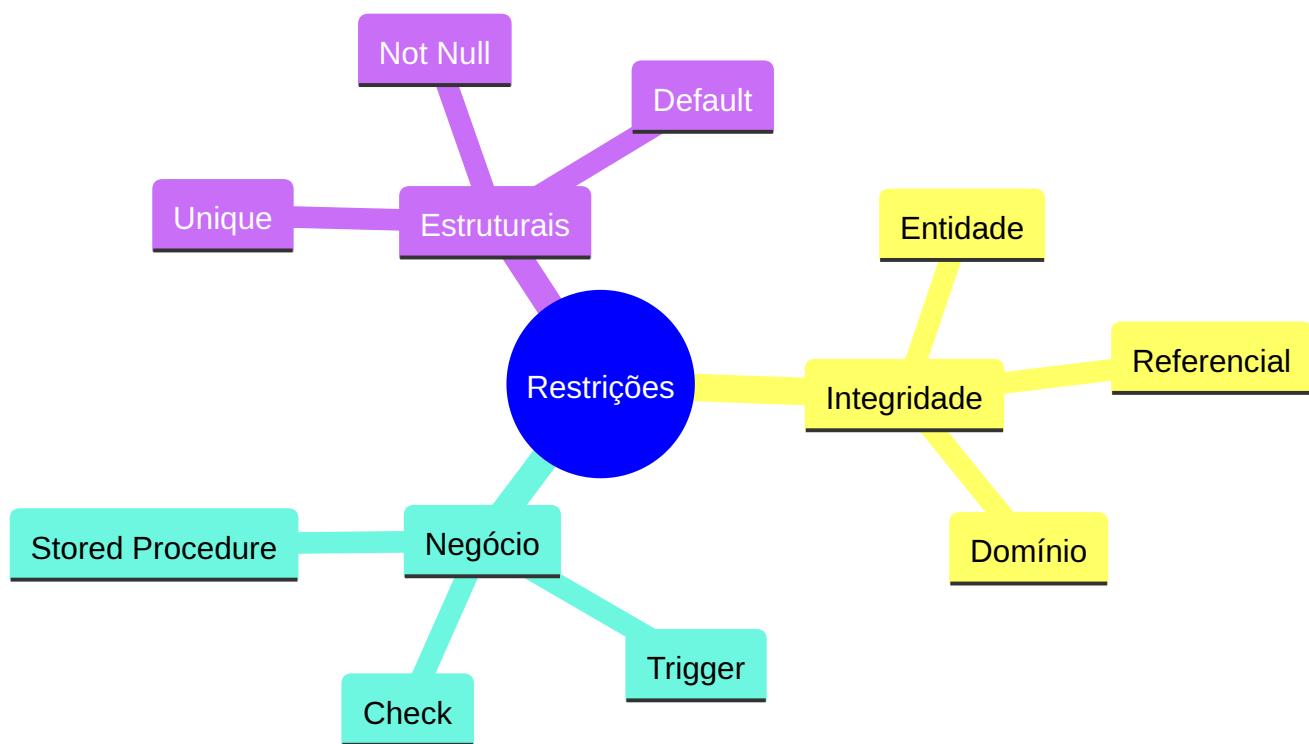
Conclusão

A escolha da estratégia de mapeamento deve considerar:

- Requisitos funcionais
- Performance necessária
- Complexidade aceitável
- Flexibilidade para evolução

Mapeamento de Restrições ER para Relacional

Visão Geral



Tipos de Restrições

1. Restrições de Chave

PRODUTO		
string	codigo	PK
string	nome	UK
string	sku	UK
decimal	preco	

Implementação

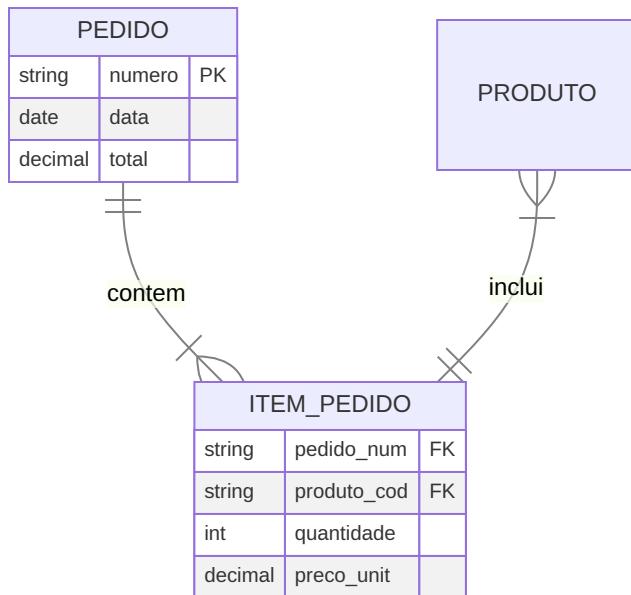
```
CREATE TABLE Produto (
    codigo VARCHAR(20) PRIMARY KEY,
```

```

        nome VARCHAR(100) NOT NULL UNIQUE,
        sku VARCHAR(50) UNIQUE,
        preco DECIMAL(10,2),
        CONSTRAINT check_preco CHECK (preco > 0)
);

```

2. Restrições de Integridade Referencial



Implementação

```

CREATE TABLE Pedido (
    numero VARCHAR(20) PRIMARY KEY,
    data DATE NOT NULL,
    total DECIMAL(10,2)
);

CREATE TABLE Item_Pedido (
    pedido_num VARCHAR(20),
    produto_cod VARCHAR(20),
    quantidade INTEGER NOT NULL,
    preco_unit DECIMAL(10,2) NOT NULL,
    PRIMARY KEY (pedido_num, produto_cod),
    FOREIGN KEY (pedido_num)
)

```

```

    REFERENCES Pedido(numero)
    ON DELETE CASCADE
    ON UPDATE CASCADE,
FOREIGN KEY (produto_cod)
    REFERENCES Produto(codigo)
    ON DELETE RESTRICT
    ON UPDATE CASCADE
);

```

3. Restrições de Domínio

FUNCIONARIO		
string	id	PK
string	nome	
string	email	
string	status	
decimal	salario	

Implementação

```

CREATE TABLE Funcionario (
    id VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) CHECK (email LIKE '%@%'),
    status VARCHAR(20) CHECK (status IN ('ATIVO', 'INATIVO',
    'FERIAS')),
    salario DECIMAL(10,2) CHECK (salario >= 0)
);

```

Restrições de Negócio

1. Validações Complexas

```

-- Trigger para validar datas
CREATE TRIGGER check_datas
BEFORE INSERT OR UPDATE ON Pedido

```

```

FOR EACH ROW
BEGIN
    IF NEW.data_entrega <= NEW.data_pedido THEN
        RAISE EXCEPTION 'Data de entrega deve ser posterior à data
do pedido';
    END IF;
END;

-- Stored Procedure para validação de estoque
CREATE PROCEDURE validar_estoque(
    p_produto_id VARCHAR,
    p_quantidade INTEGER
) AS $$

BEGIN
    IF NOT EXISTS (
        SELECT 1 FROM Estoque
        WHERE produto_id = p_produto_id
        AND quantidade_disponivel >= p_quantidade
    ) THEN
        RAISE EXCEPTION 'Estoque insuficiente';
    END IF;
END;
$$ LANGUAGE plpgsql;

```

2. Restrições Temporais

```

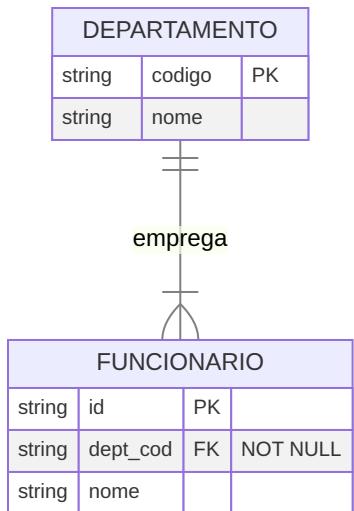
CREATE TABLE Contrato (
    id VARCHAR(20) PRIMARY KEY,
    data_inicio DATE NOT NULL,
    data_fim DATE,
    valor DECIMAL(10,2),
    CONSTRAINT check_datas
        CHECK (data_fim IS NULL OR data_fim > data_inicio),
    CONSTRAINT check_vigencia

```

```
    CHECK (data_fim IS NULL OR data_fim > CURRENT_DATE)
);
```

Mapeamento de Participação

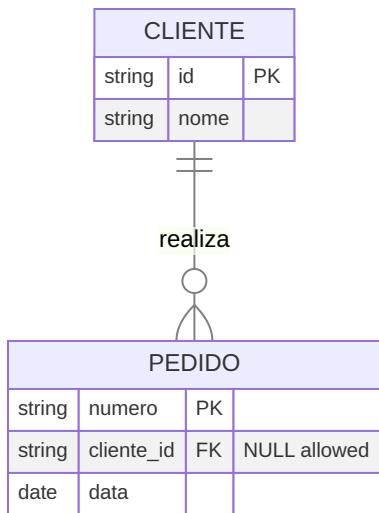
1. Participação Total



Implementação

```
CREATE TABLE Funcionario (
    id VARCHAR(20) PRIMARY KEY,
    dept_cod VARCHAR(20) NOT NULL,
    nome VARCHAR(100) NOT NULL,
    FOREIGN KEY (dept_cod)
        REFERENCES Departamento(codigo)
        ON DELETE RESTRICT
);
```

2. Participação Parcial



Implementação

```

CREATE TABLE Pedido (
    numero VARCHAR(20) PRIMARY KEY,
    cliente_id VARCHAR(20),
    data DATE NOT NULL,
    FOREIGN KEY (cliente_id)
        REFERENCES Cliente(id)
        ON DELETE SET NULL
);

```

Otimizações

1. Índices para Restrições

```

-- Índices para chaves estrangeiras
CREATE INDEX idx_item_pedido_num ON Item_Pedido(pedido_num);
CREATE INDEX idx_item_produto_cod ON Item_Pedido(produto_cod);

-- Índices para validações frequentes
CREATE INDEX idx_funcionario_status ON Funcionario(status);
CREATE INDEX idx_contrato_datas ON Contrato(data_inicio,
data_fim);

```

2. Particionamento

```
-- Particionamento por status
CREATE TABLE Pedido (
    numero VARCHAR(20),
    status VARCHAR(20),
    data DATE,
    total DECIMAL(10,2)
) PARTITION BY LIST (status);

CREATE TABLE pedido_pendente
    PARTITION OF Pedido FOR VALUES IN ('PENDENTE');
CREATE TABLE pedido_aprovado
    PARTITION OF Pedido FOR VALUES IN ('APROVADO');
CREATE TABLE pedido_cancelado
    PARTITION OF Pedido FOR VALUES IN ('CANCELADO');
```

Boas Práticas

1. Nomenclatura

- Prefixos consistentes para constraints
- Nomes descritivos para regras de negócio
- Padrão para índices e triggers

2. Documentação

```
COMMENT ON TABLE Produto IS 'Cadastro de produtos
comercializados';
COMMENT ON COLUMN Produto.codigo IS 'Código único do produto';
COMMENT ON CONSTRAINT check_preco ON Produto
    IS 'Garante que o preço seja sempre positivo';
```

3. Manutenção

- Monitoramento de violações
- Logs de alterações
- Revisão periódica

Considerações de Performance

1. Análise de Impacto

- Custo de verificações
- Frequência de validações
- Complexidade das regras

2. Estratégias de Otimização

- Uso adequado de índices
- Particionamento eficiente
- Cache de validações

Conclusão

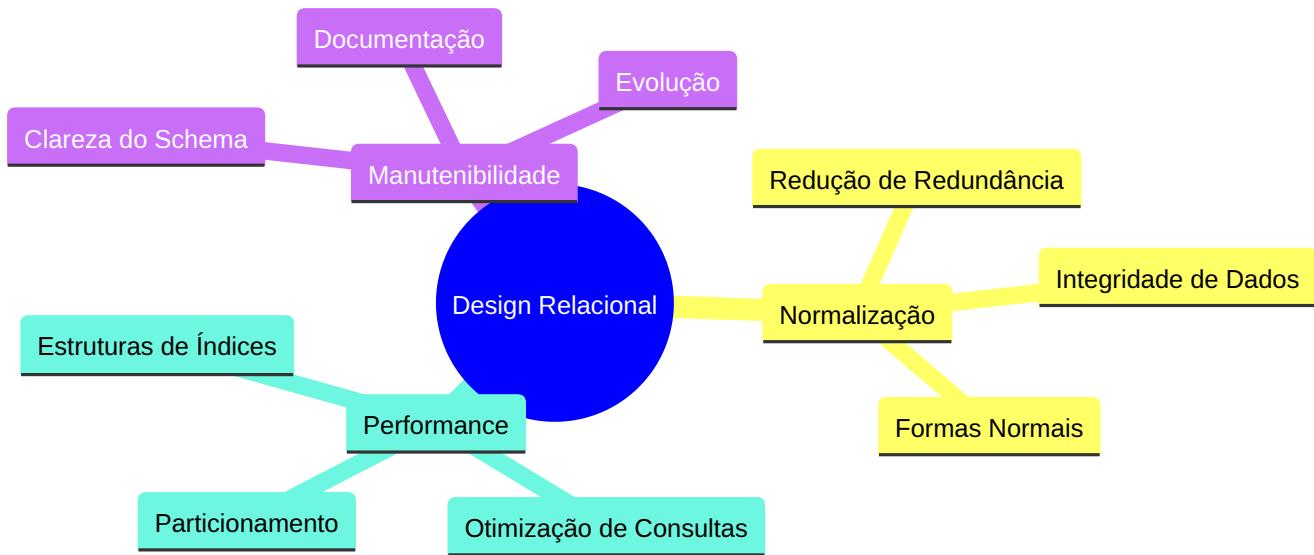
O mapeamento eficiente de restrições:

- Garante integridade dos dados
- Implementa regras de negócio
- Otimiza performance
- Facilita manutenção

Design Relacional

O design relacional é um processo sistemático para criar esquemas de banco de dados que sejam eficientes, consistentes e mantenham a integridade dos dados.

Princípios Fundamentais



Processo de Design

1. Análise de Requisitos

- Identificação de entidades e relacionamentos
- Definição de restrições
- Levantamento de requisitos de consulta
- Requisitos de performance

2. Modelagem Conceitual

- Criação do modelo ER
- Definição de cardinalidades

- Identificação de chaves
- Documentação de regras de negócio

3. Modelagem Lógica

- Transformação para modelo relacional
- Normalização de tabelas
- Definição de constraints
- Otimização inicial

4. Modelagem Física

- Escolha de tipos de dados
- Estratégia de indexação
- Particionamento
- Configurações de armazenamento

Considerações Importantes

1. Integridade dos Dados

- Restrições de domínio
- Integridade referencial
- Regras de negócio
- Validações

2. Performance

- Análise de consultas frequentes
- Estratégias de otimização

- Balanceamento de recursos
- Monitoramento

3. Escalabilidade

- Crescimento de dados
- Evolução do schema
- Particionamento
- Distribuição

Melhores Práticas

1. Nomenclatura

- Padrões consistentes
- Nomes descritivos
- Convenções estabelecidas
- Documentação clara

2. Normalização

- Nível adequado de normalização
- Casos para desnormalização
- Balanceamento com performance
- Manutenção da integridade

3. Documentação

- Dicionário de dados
- Diagramas atualizados

- Decisões de design
- Regras de negócio

Ferramentas e Técnicas

1. Modelagem

- Ferramentas CASE
- Geradores de documentação
- Validadores de schema
- Otimizadores

2. Análise

- Analisadores de performance
- Ferramentas de profiling
- Monitores de consulta
- Validadores de integridade

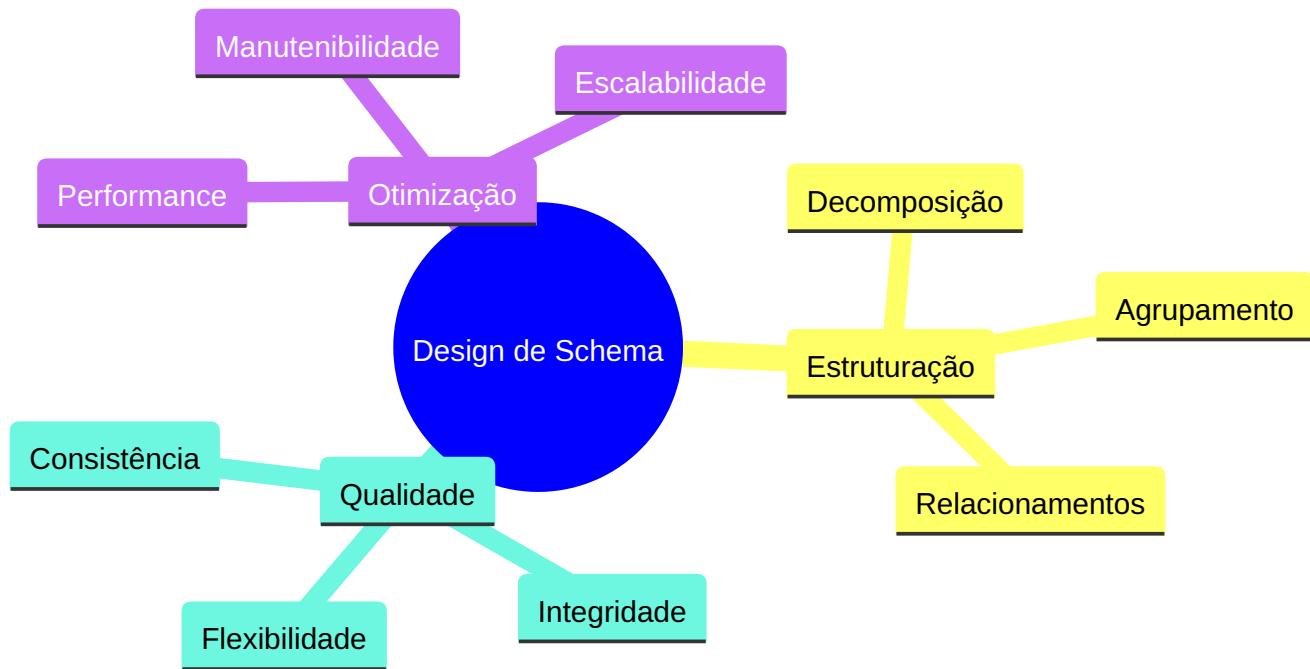
3. Manutenção

- Controle de versão
- Ferramentas de migração
- Gestão de mudanças
- Backup e recuperação

Design de Schema

O design de schema é uma etapa crucial no desenvolvimento de bancos de dados relacionais, focando na estruturação eficiente das tabelas e seus relacionamentos.

Princípios de Design



Etapas do Design

1. Análise de Requisitos

- Identificação de entidades
- Mapeamento de relacionamentos
- Definição de restrições
- Requisitos de dados

2. Estruturação Inicial

- Definição de tabelas

- Estabelecimento de chaves
- Mapeamento de relacionamentos
- Definição de constraints

3. Refinamento

- Normalização apropriada
- Otimização de estruturas
- Validação de integridade
- Ajustes de performance

Padrões de Design

1. Padrões de Chave

- Chaves naturais vs. surrogate
- Estratégias de geração
- Unicidade e integridade
- Indexação eficiente

2. Padrões de Relacionamento

- One-to-One
- One-to-Many
- Many-to-Many
- Auto-relacionamentos

3. Padrões de Dados

- Tipos apropriados

- Constraints de domínio
- Valores default
- Validações

Considerações Práticas

1. Performance

- Estruturas de índice
- Particionamento
- Clustering
- Otimização de queries

2. Manutenibilidade

- Nomenclatura clara
- Documentação adequada
- Versionamento
- Evolução do schema

3. Escalabilidade

- Crescimento de dados
- Distribuição
- Replicação
- Sharding

Antipadrões

1. Estruturais

- Redundância excessiva
- Relacionamentos circulares
- Chaves compostas complexas
- Falta de normalização

2. Implementação

- Tipos de dados inadequados
- Constraints ausentes
- Índices mal planejados
- Falta de documentação

Ferramentas e Técnicas

1. Modelagem

- Diagramas ER
- Ferramentas CASE
- Geradores de DDL
- Validadores de schema

2. Validação

- Testes de integridade
- Análise de performance
- Verificação de constraints
- Revisão de design

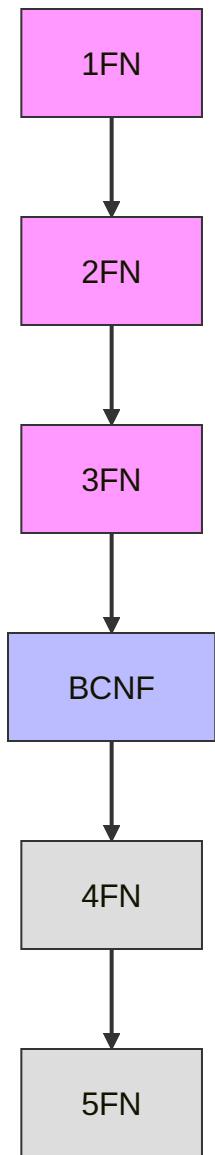
3. Manutenção

- Controle de versão
- Migrations
- Monitoramento
- Otimização contínua

Formas Normais

As formas normais são regras de design que ajudam a estruturar bancos de dados relacionais, reduzindo redundância e garantindo consistência dos dados.

Visão Geral



Primeira Forma Normal (1FN)

Regras

- Valores atômicos
- Sem grupos repetitivos
- Identificador único para cada registro

Exemplo

Antes da 1FN

```
Cliente(id, nome, telefones)
1, João Silva, "999999999, 888888888"
```

Depois da 1FN

```
Cliente(id, nome)
1, João Silva

Telefone(cliente_id, numero)
1, 999999999
1, 888888888
```

Segunda Forma Normal (2FN)

Regras

- Deve estar na 1FN
- Todos os atributos não-chave dependem totalmente da chave primária

Exemplo

Antes da 2FN

```
Pedido(cliente_id, produto_id, data_pedido, valor_produto,
nome_produto)
```

Depois da 2FN

```
Pedido(cliente_id, produto_id, data_pedido)  
Produto(id, nome, valor)
```

Terceira Forma Normal (3FN)

Regras

- Deve estar na 2FN
- Sem dependências transitivas

Exemplo

Antes da 3FN

```
Funcionario(id, nome, departamento_id, nome_departamento)
```

Depois da 3FN

```
Funcionario(id, nome, departamento_id)  
Departamento(id, nome)
```

Forma Normal de Boyce-Codd (BCNF)

Regras

- Deve estar na 3FN
- Toda dependência funcional não-trivial é determinada por uma chave candidata

Exemplo

Antes da BCNF

```
Professor_Disciplina(professor_id, disciplina, departamento)
```

Depois da BCNF

```
Professor_Departamento(professor_id, departamento)
Departamento_Disciplina(departamento, disciplina)
```

Quarta Forma Normal (4FN)

Regras

- Deve estar na BCNF
- Sem dependências multivaloradas

Exemplo

Antes da 4FN

```
Funcionario_Habilidade_Projeto(func_id, habilidade, projeto)
```

Depois da 4FN

```
Funcionario_Habilidade(func_id, habilidade)
Funcionario_Projeto(func_id, projeto)
```

Quinta Forma Normal (5FN)

Regras

- Deve estar na 4FN
- Sem dependências de junção

Considerações Práticas

- Raramente necessária
- Complexidade elevada
- Casos específicos

Desnormalização

Quando Considerar

- Performance crítica
- Dados predominantemente estáticos
- Consultas complexas frequentes
- Requisitos específicos de negócio

Riscos

- Redundância de dados
- Anomalias de atualização
- Complexidade de manutenção
- Inconsistência potencial

Recomendações

1. Análise de Requisitos

- Padrões de acesso
- Volume de dados
- Frequência de atualizações
- Requisitos de performance

2. Balanceamento

- Normalização vs. Performance
- Complexidade vs. Simplicidade

- Flexibilidade vs. Otimização
- Manutenibilidade vs. Eficiência

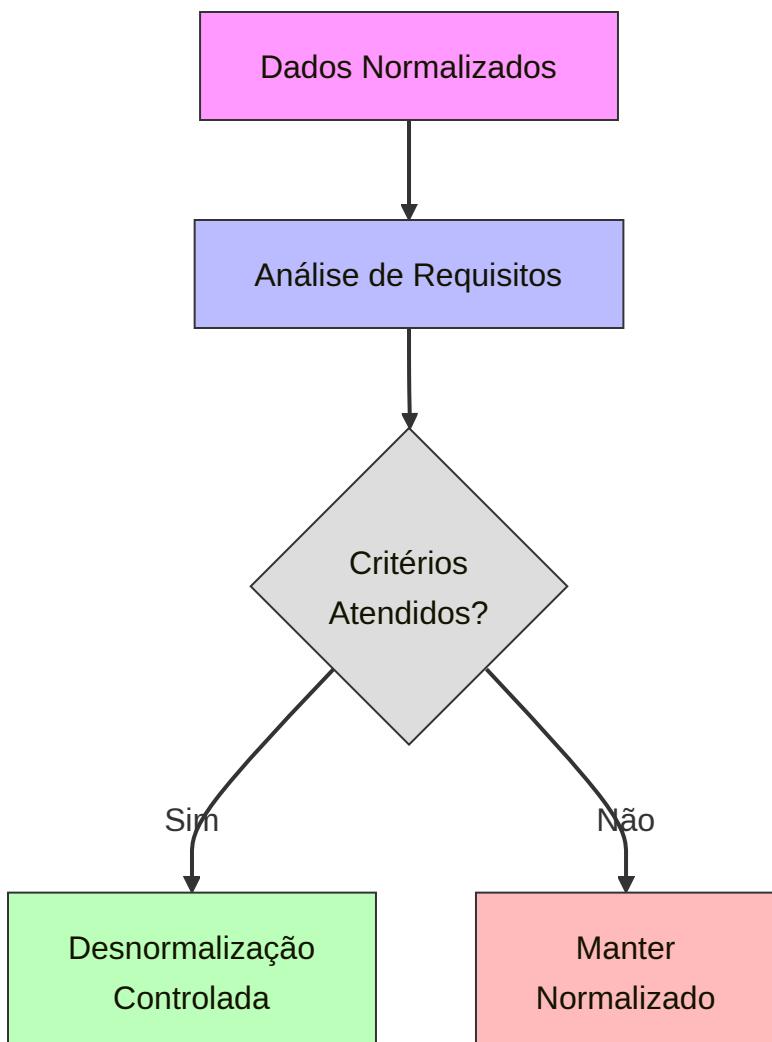
3. Documentação

- Decisões de design
- Exceções à normalização
- Justificativas
- Impactos e trade-offs

Intro Desnormalização

A desnormalização é uma estratégia de design de banco de dados que introduz redundância controlada para melhorar a performance em casos específicos.

Conceito



Quando Desnormalizar

1. Performance Crítica

- Consultas complexas frequentes
- Joins custosos

- Requisitos de tempo real
- Alta carga de leitura

2. Dados Estáticos

- Baixa frequência de atualizações
- Dados históricos
- Dados de referência
- Informações catalográficas

3. Requisitos Específicos

- Análises estatísticas
- Relatórios complexos
- Agregações frequentes
- Cache de dados

Técnicas de Desnормalização

1. Duplicação de Dados

```
-- Normalizado
Cliente(id, nome)
Pedido(id, cliente_id, data)

-- Desnormalizado
Pedido(id, cliente_id, cliente_nome, data)
```

2. Tabelas Agregadas

```
-- Normalizado
Venda(id, produto_id, quantidade, valor)
```

```
-- Desnормalizado  
Venda_Diaria(data, total_vendas, total_valor)
```

3. Campos Calculados

```
-- Normalizado  
Produto(id, preco)  
Item_Pedido(pedido_id, produto_id, quantidade)  
  
-- Desnornalizado  
Item_Pedido(pedido_id, produto_id, quantidade, valor_total)
```

Riscos e Desafios

1. Integridade de Dados

- Inconsistências potenciais
- Complexidade de atualizações
- Sincronização de dados
- Validação adicional

2. Manutenção

- Código mais complexo
- Maior espaço em disco
- Processos de atualização
- Documentação necessária

3. Performance

- Overhead em escritas

- Custos de storage
- Backup e recuperação
- Índices adicionais

Estratégias de Implementação

1. Análise Prévia

- Perfil de carga
- Padrões de acesso
- Requisitos de consistência
- Custos vs. benefícios

2. Implementação Controlada

- Mudanças incrementais
- Testes de performance
- Monitoramento
- Rollback plan

3. Manutenção

- Processos de sincronização
- Verificações periódicas
- Ajustes de performance
- Documentação atualizada

Melhores Práticas

1. Documentação

- Justificativas
- Impactos
- Dependências
- Procedimentos

2. Monitoramento

- Performance metrics
- Uso de storage
- Consistência de dados
- Logs de atualização

3. Revisão Periódica

- Validação de benefícios
- Ajustes necessários
- Evolução do sistema
- Reavaliação de decisões

Exemplos Práticos

1. E-Commerce

```
-- Antes
Produto(id, nome, preco)
Categoria(id, nome)
Produto_Categoria(produto_id, categoria_id)
```

```
-- Depois  
Produto(id, nome, preco, categoria_nome)
```

2. Sistema de Relatórios

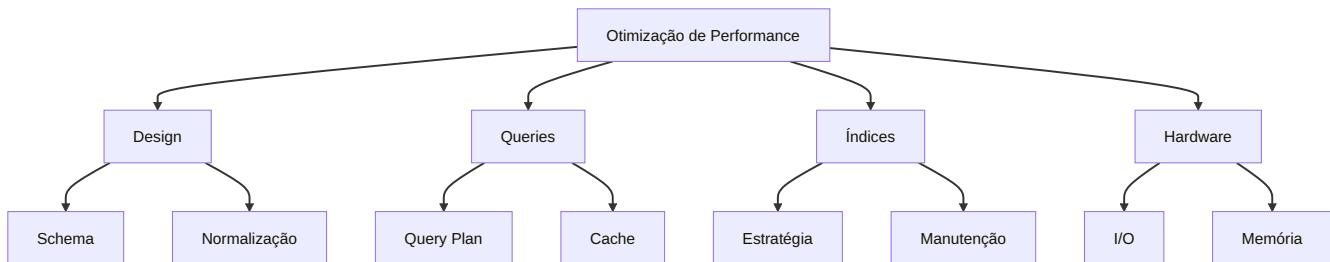
```
-- Antes  
Venda(id, data, valor)  
  
-- Depois  
Venda(id, data, valor)  
Venda_Mensal(ano, mes, total_vendas, valor_total)
```

3. Gestão de Conteúdo

```
-- Antes  
Artigo(id, titulo, conteudo)  
Tag(id, nome)  
Artigo_Tag(artigo_id, tag_id)  
  
-- Depois  
Artigo(id, titulo, conteudo, tags_concatenadas)
```

Otimização de Performance em Bancos de Dados

Visão Geral



Estratégias de Otimização

1. Design de Schema

- Normalização apropriada
- Tipos de dados eficientes
- Particionamento
- Clustering

2. Otimização de Queries

Análise de Plano de Execução

```
EXPLAIN ANALYZE
SELECT *
FROM pedidos p
JOIN clientes c ON p.cliente_id = c.id
WHERE p.status = 'PENDENTE';
```

Técnicas Comuns

- Minimizar SELECT *

- Usar JOINs eficientes
- Evitar subqueries desnecessárias
- Utilizar índices apropriadamente

3. Indexação

Estratégias

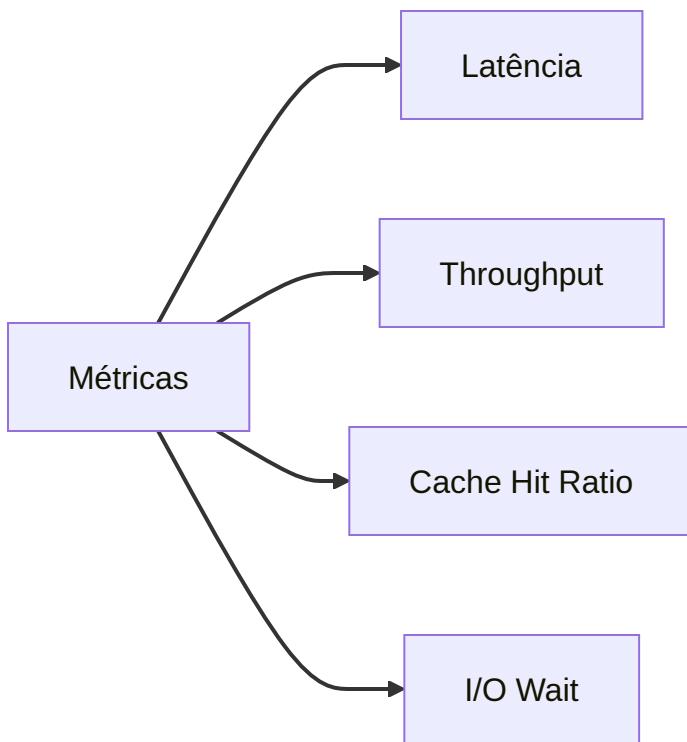
- Índices compostos
- Índices parciais
- Índices cobertos
- Manutenção regular

```
-- Índice composto para queries frequentes
CREATE INDEX idx_pedidos_cliente_data ON pedidos(cliente_id,
data_pedido);

-- Índice parcial para filtros comuns
CREATE INDEX idx_pedidos_pendentes ON pedidos(data_pedido)
WHERE status = 'PENDENTE';
```

Monitoramento e Análise

1. Métricas Principais



2. Ferramentas de Análise

- Query analyzers
- Profilers
- Monitoring dashboards
- Log analysis

Técnicas Avançadas

1. Particionamento

```

CREATE TABLE vendas (
    id SERIAL,
    data_venda DATE,
    valor DECIMAL
) PARTITION BY RANGE (data_venda);
  
```

```
CREATE TABLE vendas_2023 PARTITION OF vendas  
FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');
```

2. Materialização

```
CREATE MATERIALIZED VIEW mv_resumo_vendas AS  
SELECT  
    data_venda::DATE,  
    SUM(valor) as total_vendas  
FROM vendas  
GROUP BY data_venda::DATE  
WITH DATA;
```

3. Caching

- Query cache
- Buffer pool
- Application-level cache
- Distributed cache

Otimizações Específicas

1. OLTP (Online Transaction Processing)

- Índices precisos
- Normalização adequada
- Transações otimizadas
- Connection pooling

2. OLAP (Online Analytical Processing)

- Desnormalização estratégica

- Agregações pré-calculadas
- Particionamento efetivo
- Parallel query

Boas Práticas

1. Design

- Escolha apropriada de tipos
- Constraints adequadas
- Normalização balanceada
- Documentação clara

2. Desenvolvimento

```
-- Evitar
SELECT * FROM usuarios WHERE nome LIKE '%João%';

-- Preferir
SELECT id, nome, email
FROM usuarios
WHERE nome_normalizado = normalize('João');
```

3. Manutenção

- Atualização de estatísticas
- Rebuild de índices
- Vacuum regular
- Monitoramento contínuo

Checklist de Otimização

1. Análise Inicial

- [] Identificar queries lentas
- [] Analisar planos de execução
- [] Verificar índices existentes
- [] Avaliar estatísticas

2. Implementação

- [] Criar/ajustar índices
- [] Otimizar queries
- [] Configurar partições
- [] Ajustar parâmetros

3. Validação

- [] Testar performance
- [] Monitorar recursos
- [] Verificar impactos
- [] Documentar mudanças

Considerações de Escalabilidade

1. Vertical Scaling

- CPU
- Memória
- Storage
- I/O

2. Horizontal Scaling

- Sharding
- Read replicas
- Load balancing
- Distributed caching

Anti-Patterns e Soluções

1. Problemas Comuns

```
-- Anti-pattern: N+1 queries
SELECT * FROM pedidos;
SELECT * FROM itens WHERE pedido_id = ?; -- Repetido N vezes

-- Solução: JOIN adequado
SELECT p.*, i.*
FROM pedidos p
JOIN itens i ON p.id = i.pedido_id;
```

2. Mitigações

- Query batching
- Eager loading
- Caching estratégico
- Query optimization

Recursos Adicionais

1. Ferramentas

- Query analyzers

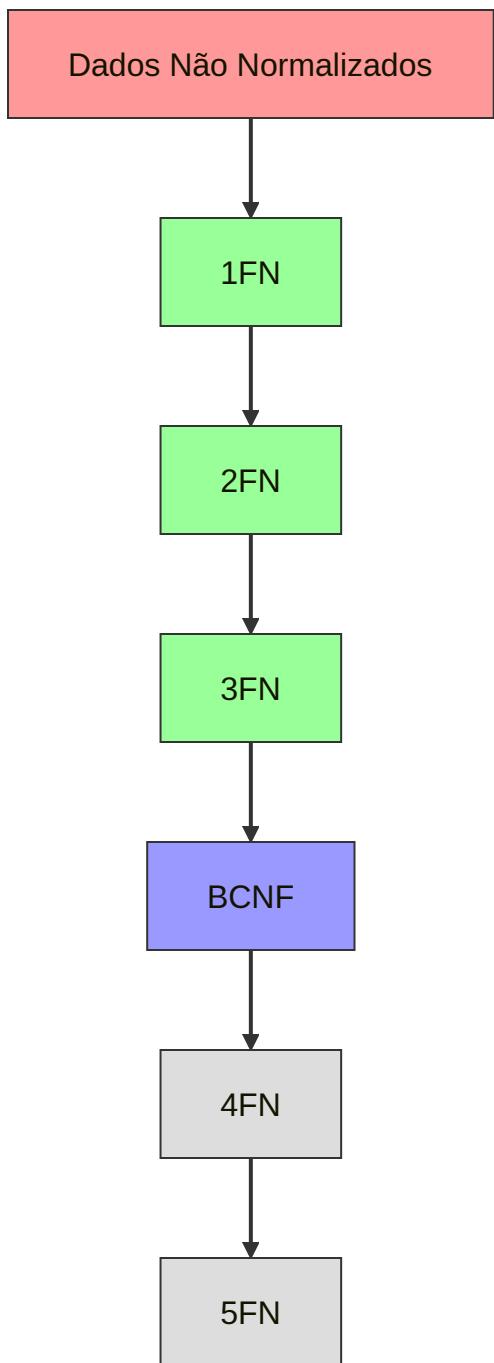
- Profiling tools
- Monitoring solutions
- Benchmarking utilities

2. Documentação

- Performance guides
- Best practices
- Case studies
- Troubleshooting guides

Normalização de Bancos de Dados

Visão Geral



Conceitos Fundamentais

1. Objetivos da Normalização

- Eliminar redundância
- Garantir consistência
- Facilitar manutenção
- Reduzir anomalias

2. Dependências Funcionais

- Dependência total
- Dependência parcial
- Dependência transitiva
- Dependência multivalorada

Formas Normais

1. Primeira Forma Normal (1FN)

```
-- Antes da 1FN
Contato(id, nome, telefones)
1, "João Silva", "999999999, 888888888"

-- Depois da 1FN
Contato(id, nome)
1, "João Silva"

Telefone(contato_id, numero)
1, "999999999"
1, "888888888"
```

2. Segunda Forma Normal (2FN)

```
-- Antes da 2FN
Pedido(cliente_id, produto_id, data_pedido, valor_produto,
nome_produto)

-- Depois da 2FN
Pedido(cliente_id, produto_id, data_pedido)
Produto(id, nome, valor)
```

3. Terceira Forma Normal (3FN)

```
-- Antes da 3FN
Funcionario(id, nome, departamento_id, nome_departamento)

-- Depois da 3FN
Funcionario(id, nome, departamento_id)
Departamento(id, nome)
```

4. Forma Normal de Boyce-Codd (BCNF)

```
-- Antes da BCNF
Professor_Disciplina(professor_id, disciplina, departamento)

-- Depois da BCNF
Professor_Departamento(professor_id, departamento)
Departamento_Disciplina(departamento, disciplina)
```

5. Quarta Forma Normal (4FN)

```
-- Antes da 4FN
Funcionario_Habilidade_Projeto(func_id, habilidade, projeto)

-- Depois da 4FN
Funcionario_Habilidade(func_id, habilidade)
Funcionario_Projeto(func_id, projeto)
```

Processo de Normalização

1. Análise de Requisitos

- Identificar entidades
- Mapear relacionamentos
- Definir atributos
- Estabelecer dependências

2. Aplicação Progressiva



3. Validação

- Testes de integridade
- Verificação de anomalias
- Análise de performance
- Revisão de requisitos

Benefícios e Considerações

1. Vantagens

- Integridade de dados
- Consistência
- Facilidade de manutenção
- Flexibilidade

2. Desvantagens

- Complexidade de queries
- Overhead de joins
- Performance em leituras
- Complexidade de implementação

Exemplos Práticos

1. Sistema de Vendas

```
-- Não normalizado
Venda(id, data, cliente_nome, cliente_email, produto_nome,
quantidade, preco)

-- Normalizado
Cliente(id, nome, email)
Produto(id, nome, preco)
Venda(id, cliente_id)
ItemVenda(venda_id, produto_id, quantidade)
```

2. Sistema Acadêmico

```
-- Não normalizado
Matricula(aluno_nome, curso_nome, disciplina_nome, professor_nome,
nota)

-- Normalizado
Aluno(id, nome)
Curso(id, nome)
Disciplina(id, nome, curso_id)
Professor(id, nome)
Matricula(aluno_id, disciplina_id, professor_id, nota)
```

Melhores Práticas

1. Design

- Começar com modelo completo
- Normalizar progressivamente
- Documentar decisões
- Manter consistência

2. Implementação

- Usar ferramentas adequadas
- Seguir padrões
- Manter rastreabilidade
- Validar continuamente

3. Manutenção

- Monitorar performance
- Ajustar quando necessário
- Manter documentação
- Revisar periodicamente

Ferramentas e Recursos

1. Design

- Modelagem ER
- CASE tools
- Diagramas UML
- Documentação

2. Validação

- Scripts de teste
- Ferramentas de análise
- Verificadores de dependência
- Analisadores de schema

Conclusão

A normalização é um processo fundamental para:

- Garantir qualidade dos dados
- Facilitar manutenção
- Reduzir redundância
- Promover consistência

Deve ser aplicada considerando:

- Requisitos do sistema
- Performance necessária
- Complexidade aceitável
- Recursos disponíveis

Primeira Forma Normal (1FN)

Definição

A Primeira Forma Normal (1FN) é o nível inicial de normalização de banco de dados que estabelece duas regras fundamentais:

1. Atomicidade dos valores
2. Eliminação de grupos repetitivos

Regras Detalhadas

1. Atomicidade

- Cada coluna deve conter valores atômicos (indivisíveis)
- Não permitir múltiplos valores em uma única célula
- Não permitir arrays ou listas como valores

2. Grupos Repetitivos

- Eliminar colunas que contêm o mesmo tipo de informação
- Criar novas tabelas para grupos de dados repetitivos
- Estabelecer relacionamentos através de chaves

Exemplos Práticos

Exemplo 1: Dados de Contato

Violação da 1FN

```
-- Tabela não normalizada
Cliente(
    id INT,
```

```
        nome VARCHAR(100),
        telefones VARCHAR(200)    -- "9999999999, 888888888"
    )
```

Aplicação da 1FN

```
-- Tabelas normalizadas
Cliente(
    id INT PRIMARY KEY,
    nome VARCHAR(100)
)

Telefone(
    cliente_id INT,
    numero VARCHAR(20),
    FOREIGN KEY (cliente_id) REFERENCES Cliente(id)
)
```

Exemplo 2: Endereços

Violação da 1FN

```
-- Tabela não normalizada
Funcionario(
    id INT,
    nome VARCHAR(100),
    endereco VARCHAR(500)    -- "Rua A, 123, São Paulo; Rua B, 456,
    Rio de Janeiro"
)
```

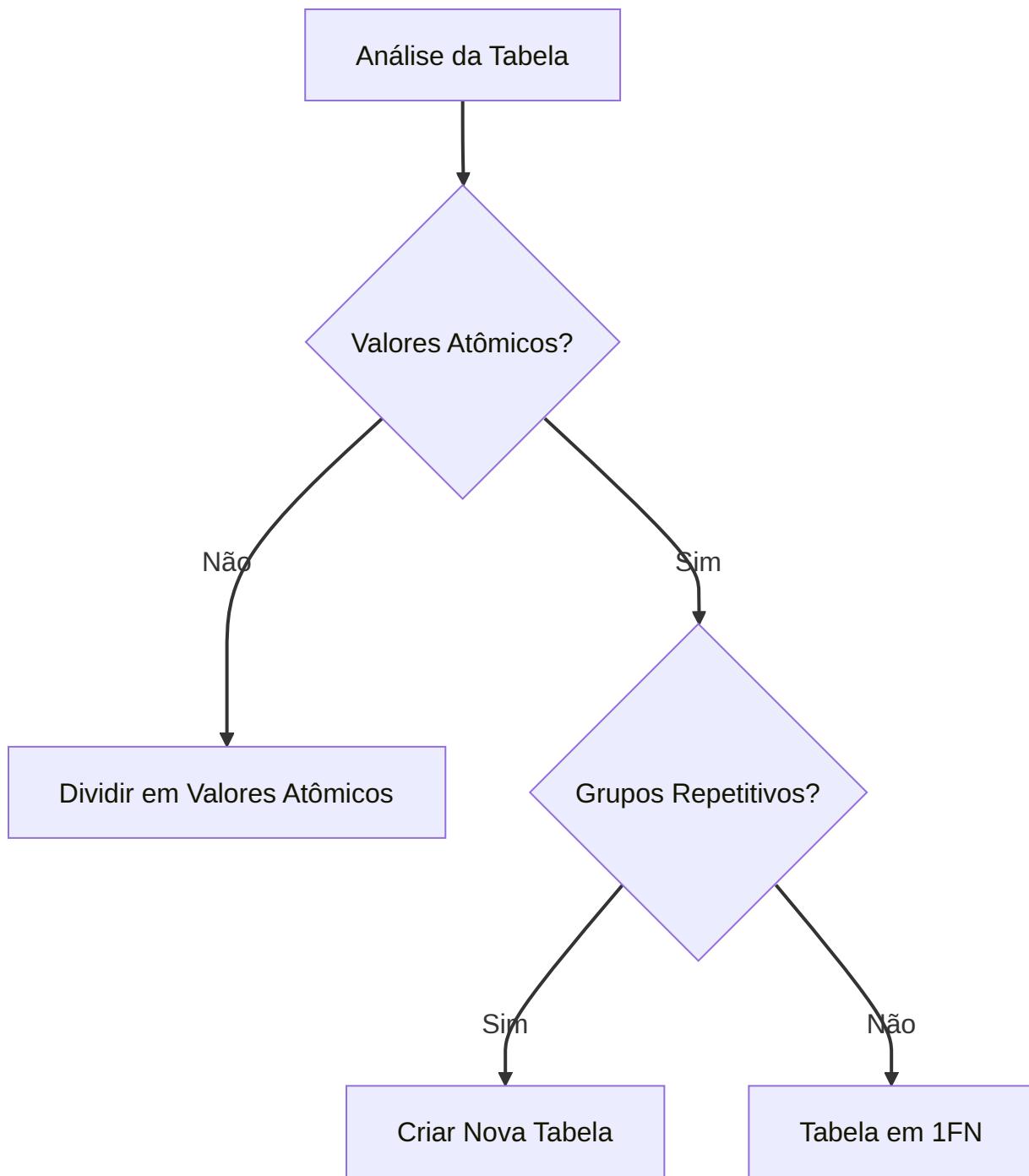
Aplicação da 1FN

```
-- Tabelas normalizadas
Funcionario(
    id INT PRIMARY KEY,
    nome VARCHAR(100)
)
```

```
Endereco(
    funcionario_id INT,
    rua VARCHAR(100),
    numero VARCHAR(10),
    cidade VARCHAR(100),
    FOREIGN KEY (funcionario_id) REFERENCES Funcionario(id)
)
```

Processo de Normalização

1. Identificação de Violações



2. Passos para Normalização

1. Identificar colunas com múltiplos valores
2. Criar novas tabelas para dados repetitivos
3. Estabelecer relacionamentos

4. Validar atomicidade

Benefícios

1. Integridade dos Dados

- Valores consistentes
- Busca facilitada
- Manipulação simplificada

2. Manutenção

- Atualizações mais simples
- Menor redundância
- Maior consistência

Considerações Práticas

1. Performance

- Aumento no número de joins
- Mais tabelas para gerenciar
- Possível impacto em consultas complexas

2. Implementação

```
-- Exemplo de implementação prática
CREATE TABLE Cliente (
    id INT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL
);
```

```
CREATE TABLE Telefone (
```

```

    id INT PRIMARY KEY,
    cliente_id INT NOT NULL,
    numero VARCHAR(20) NOT NULL,
    tipo VARCHAR(20),
    FOREIGN KEY (cliente_id) REFERENCES Cliente(id)
);

-- Inserção de dados
INSERT INTO Cliente (id, nome) VALUES (1, 'João Silva');

INSERT INTO Telefone (id, cliente_id, numero, tipo) VALUES
(1, 1, '9999999999', 'Celular'),
(2, 1, '8888888888', 'Residencial');

```

Checklist de Validação

1. Verificação de Conformidade

- [] Todos os valores são atômicos?
- [] Não existem grupos repetitivos?
- [] Chaves primárias definidas?
- [] Relacionamentos estabelecidos?

2. Testes

- [] Inserção de dados
- [] Atualização de registros
- [] Exclusão de registros
- [] Consultas básicas

Anti-Padrões Comuns

1. Violações Frequentes

```
-- Anti-padrão: Valores múltiplos em uma coluna
CREATE TABLE Produto (
    id INT PRIMARY KEY,
    nome VARCHAR(100),
    categorias VARCHAR(500) -- "Eletrônicos, Informática,
Acessórios"
);

-- Correção: Tabela separada para categorias
CREATE TABLE Categoria (
    produto_id INT,
    categoria VARCHAR(100),
    FOREIGN KEY (produto_id) REFERENCES Produto(id)
);
```

2. Soluções

- Identificar e corrigir valores não atômicos
- Normalizar grupos repetitivos
- Estabelecer relacionamentos adequados

Conclusão

A Primeira Forma Normal é fundamental para:

- Garantir a integridade dos dados
- Facilitar a manutenção
- Estabelecer base para outras formas normais
- Promover boas práticas de modelagem

Deve ser aplicada considerando:

- Requisitos do sistema
- Necessidades de performance
- Complexidade aceitável
- Facilidade de manutenção

Segunda Forma Normal (2FN)

Definição

A Segunda Forma Normal (2FN) é um nível de normalização que exige:

1. A tabela deve estar na 1FN
2. Todos os atributos não-chave devem depender totalmente da chave primária

Conceitos Fundamentais

1. Dependência Funcional Total

- Todos os atributos não-chave dependem da chave primária completa
- Não existem dependências parciais
- Aplicável principalmente em chaves compostas

2. Dependência Funcional Parcial

- Quando um atributo depende apenas de parte da chave primária
- Deve ser eliminada para atingir a 2FN
- Comum em tabelas com chaves compostas

Exemplos Práticos

Exemplo 1: Pedidos

Violação da 2FN

```
-- Tabela não normalizada
CREATE TABLE Pedido (
    cliente_id INT,
    produto_id INT,
```

```
    data_pedido DATE,  
    quantidade INT,  
    valor_produto DECIMAL(10,2),  
    nome_produto VARCHAR(100),  
    PRIMARY KEY (cliente_id, produto_id)  
);
```

Aplicação da 2FN

```
-- Tabelas normalizadas  
CREATE TABLE Pedido (  
    cliente_id INT,  
    produto_id INT,  
    data_pedido DATE,  
    quantidade INT,  
    PRIMARY KEY (cliente_id, produto_id)  
);  
  
CREATE TABLE Produto (  
    id INT PRIMARY KEY,  
    nome VARCHAR(100),  
    valor DECIMAL(10,2)  
);
```

Exemplo 2: Cursos e Professores

Violação da 2FN

```
-- Tabela não normalizada  
CREATE TABLE Curso_Professor (  
    curso_id INT,  
    professor_id INT,  
    nome_curso VARCHAR(100),  
    departamento_curso VARCHAR(50),  
    nome_professor VARCHAR(100),
```

```
    PRIMARY KEY (curso_id, professor_id)
);
```

Aplicação da 2FN

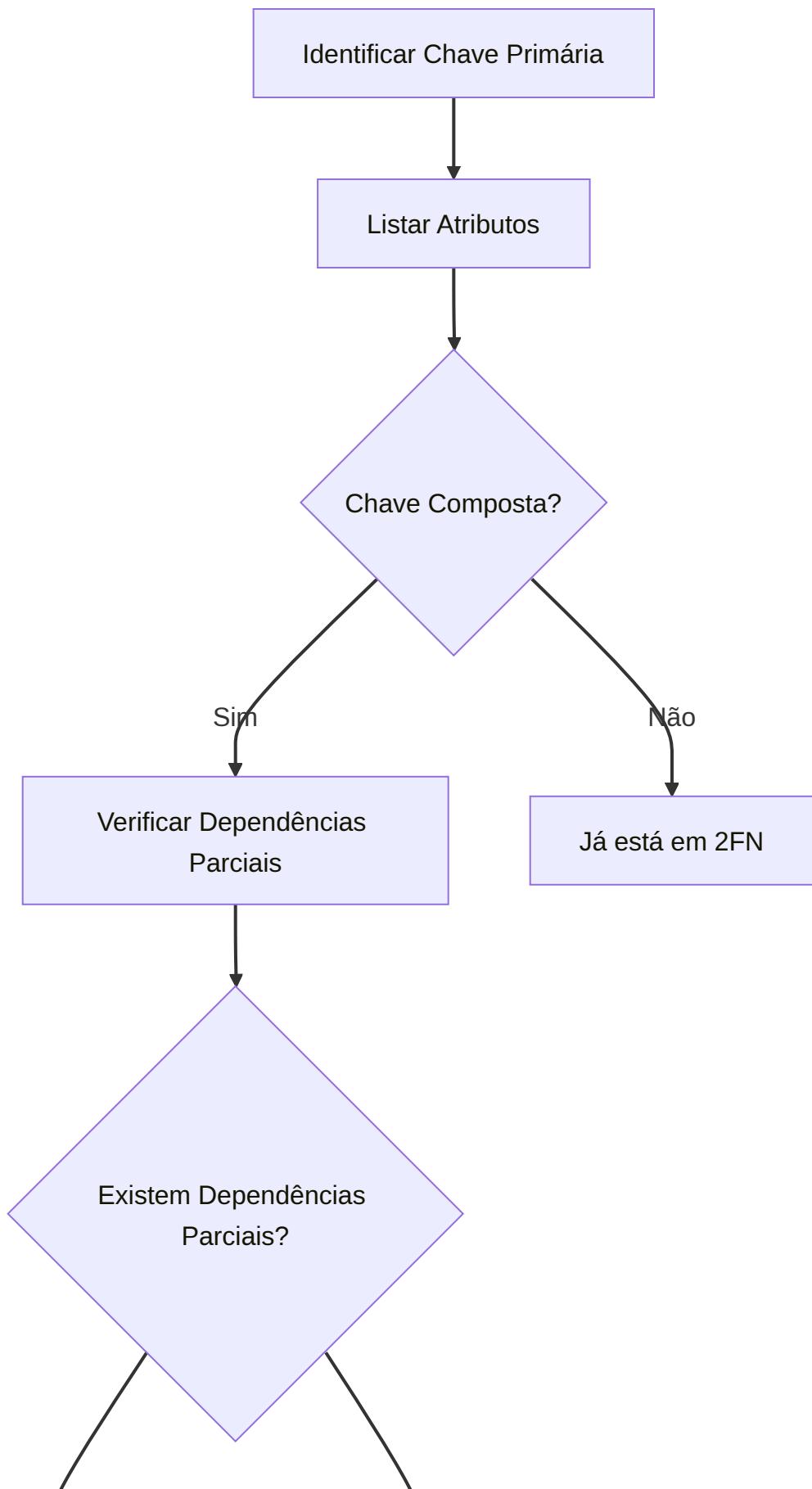
```
-- Tabelas normalizadas
CREATE TABLE Curso (
    id INT PRIMARY KEY,
    nome VARCHAR(100),
    departamento VARCHAR(50)
);

CREATE TABLE Curso_Professor (
    curso_id INT,
    professor_id INT,
    PRIMARY KEY (curso_id, professor_id),
    FOREIGN KEY (curso_id) REFERENCES Curso(id),
    FOREIGN KEY (professor_id) REFERENCES Professor(id)
);

CREATE TABLE Professor (
    id INT PRIMARY KEY,
    nome VARCHAR(100)
);
```

Processo de Identificação

1. Análise de Dependências





2. Passos para Normalização

1. Identificar a chave primária
2. Verificar dependências funcionais
3. Separar atributos com dependência parcial
4. Criar novas tabelas quando necessário

Benefícios

1. Estrutura de Dados

- Eliminação de redundância
- Melhor organização
- Maior consistência

2. Integridade

- Dados mais confiáveis
- Atualizações mais seguras
- Menor risco de anomalias

Considerações Práticas

1. Implementação

```
-- Exemplo de migração de dados
INSERT INTO Produto (id, nome, valor)
```

```
SELECT DISTINCT produto_id, nome_produto, valor_produto
FROM pedido_antigo;

INSERT INTO Pedido (cliente_id, produto_id, data_pedido,
quantidade)
SELECT cliente_id, produto_id, data_pedido, quantidade
FROM pedido_antigo;
```

2. Performance

- Avaliação de impacto
- Balanceamento de normalização
- Considerações de consulta

Checklist de Validação

1. Verificação

- [] Tabela está em 1FN?
- [] Chave primária identificada?
- [] Dependências funcionais mapeadas?
- [] Dependências parciais eliminadas?

2. Testes

- [] Integridade dos dados
- [] Consistência das relações
- [] Performance das consultas
- [] Facilidade de manutenção

Anti-Padrões

1. Violações Comuns

```
-- Anti-padrão: Dependência parcial
CREATE TABLE Inscricao_Curso (
    aluno_id INT,
    curso_id INT,
    data_inscricao DATE,
    nome_curso VARCHAR(100), -- Depende apenas de curso_id
    PRIMARY KEY (aluno_id, curso_id)
);

-- Correção
CREATE TABLE Curso (
    id INT PRIMARY KEY,
    nome VARCHAR(100)
);

CREATE TABLE Inscricao (
    aluno_id INT,
    curso_id INT,
    data_inscricao DATE,
    PRIMARY KEY (aluno_id, curso_id),
    FOREIGN KEY (curso_id) REFERENCES Curso(id)
);
```

2. Soluções

- Identificar dependências parciais
- Criar tabelas separadas
- Estabelecer relacionamentos apropriados
- Manter documentação atualizada

Conclusão

A Segunda Forma Normal é essencial para:

- Eliminar redundâncias
- Melhorar a organização dos dados
- Facilitar a manutenção
- Garantir consistência

Deve ser implementada considerando:

- Requisitos do sistema
- Complexidade das consultas
- Necessidades de performance
- Facilidade de manutenção

Terceira Forma Normal (3FN)

Definição

A Terceira Forma Normal (3FN) é um nível de normalização que exige:

1. A tabela deve estar na 2FN
2. Não deve haver dependências transitivas entre atributos não-chave

Conceitos Fundamentais

1. Dependência Transitiva

- Ocorre quando um atributo não-chave depende de outro atributo não-chave
- $A \rightarrow B$ e $B \rightarrow C$, então $A \rightarrow C$ (transitividade)
- Deve ser eliminada para atingir a 3FN

2. Identificação de Dependências

- Mapeamento de todas as dependências funcionais
- Análise de relacionamentos indiretos
- Identificação de atributos determinantes

Exemplos Práticos

Exemplo 1: Funcionários e Departamentos

Violação da 3FN

```
-- Tabela não normalizada
CREATE TABLE Funcionario (
    id INT PRIMARY KEY,
    nome VARCHAR(100),
```

```
departamento_id INT,  
nome_departamento VARCHAR(100),  
localizacao_departamento VARCHAR(100)  
);
```

Aplicação da 3FN

```
-- Tabelas normalizadas  
CREATE TABLE Funcionario (  
    id INT PRIMARY KEY,  
    nome VARCHAR(100),  
    departamento_id INT,  
    FOREIGN KEY (departamento_id) REFERENCES Departamento(id)  
);  
  
CREATE TABLE Departamento (  
    id INT PRIMARY KEY,  
    nome VARCHAR(100),  
    localizacao VARCHAR(100)  
);
```

Exemplo 2: Pedidos e Clientes

Violação da 3FN

```
-- Tabela não normalizada  
CREATE TABLE Pedido (  
    id INT PRIMARY KEY,  
    cliente_id INT,  
    nome_cliente VARCHAR(100),  
    cidade_cliente VARCHAR(100),  
    estado_cliente VARCHAR(2)  
);
```

Aplicação da 3FN

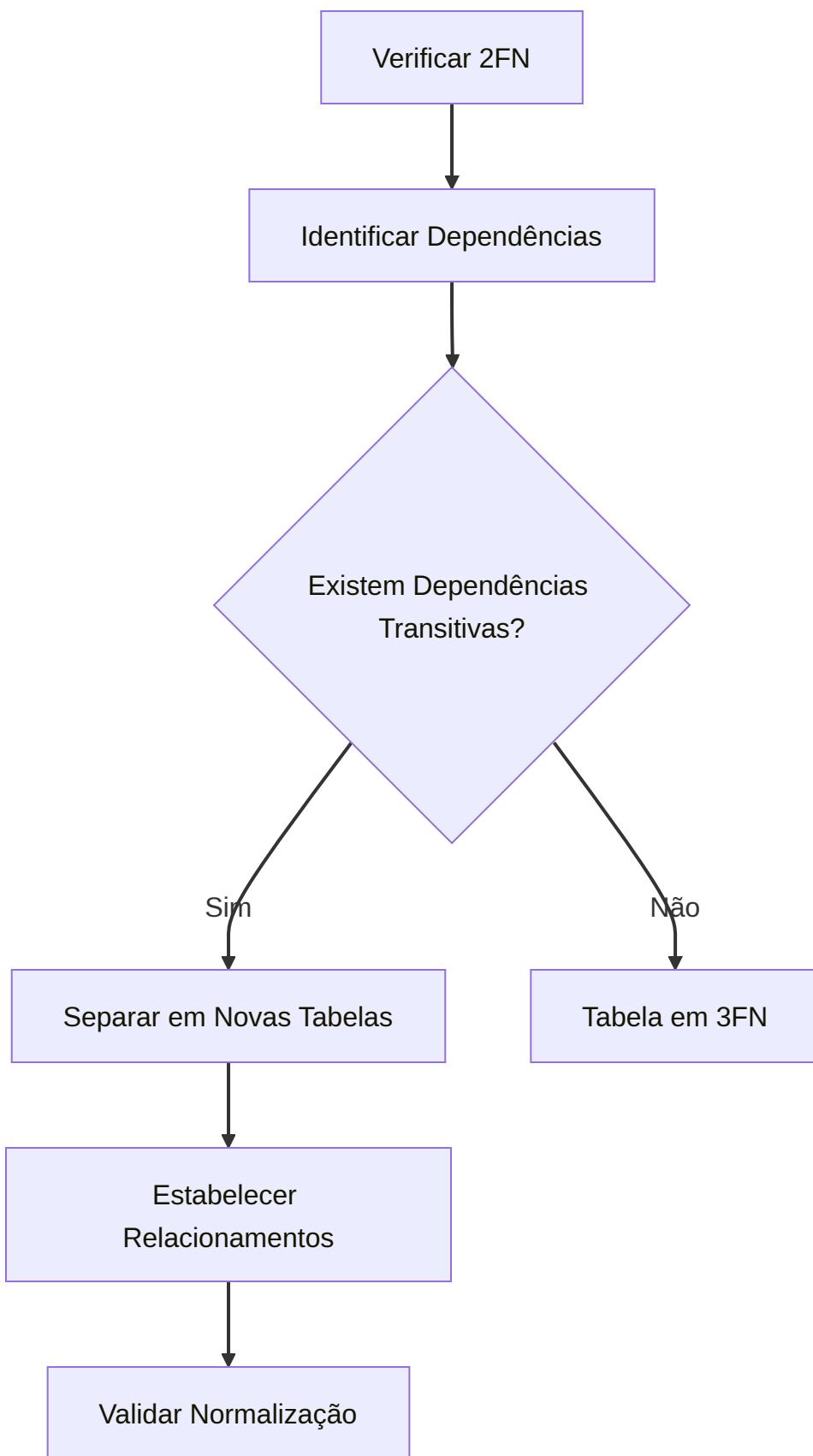
```
-- Tabelas normalizadas
CREATE TABLE Pedido (
    id INT PRIMARY KEY,
    cliente_id INT,
    FOREIGN KEY (cliente_id) REFERENCES Cliente(id)
);

CREATE TABLE Cliente (
    id INT PRIMARY KEY,
    nome VARCHAR(100),
    cidade_id INT,
    FOREIGN KEY (cidade_id) REFERENCES Cidade(id)
);

CREATE TABLE Cidade (
    id INT PRIMARY KEY,
    nome VARCHAR(100),
    estado VARCHAR(2)
);
```

Processo de Normalização

1. Identificação de Dependências



2. Passos para Normalização

1. Confirmar 2FN
2. Identificar dependências transitivas
3. Criar novas tabelas
4. Estabelecer relacionamentos
5. Validar integridade

Benefícios

1. Qualidade dos Dados

- Eliminação de redundância
- Maior consistência
- Integridade referencial

2. Manutenção

- Atualizações simplificadas
- Menor risco de anomalias
- Melhor organização

Considerações Práticas

1. Implementação

```
-- Exemplo de migração de dados
INSERT INTO Departamento (id, nome, localizacao)
SELECT DISTINCT departamento_id, nome_departamento,
localizacao_departamento
FROM funcionario_antigo;

UPDATE Funcionario f
SET departamento_id = (
```

```
SELECT d.id  
FROM Departamento d  
WHERE d.nome = f.nome_departamento  
);
```

2. Performance

- Análise de impacto
- Otimização de consultas
- Índices adequados

Checklist de Validação

1. Verificação

- [] Tabela está em 2FN?
- [] Dependências transitivas identificadas?
- [] Novas tabelas criadas corretamente?
- [] Relacionamentos estabelecidos?

2. Testes

- [] Integridade dos dados
- [] Consultas otimizadas
- [] Atualizações funcionais
- [] Performance adequada

Anti-Padrões

1. Violações Comuns

```

-- Anti-padrão: Dependência transitiva
CREATE TABLE Venda (
    id INT PRIMARY KEY,
    vendedor_id INT,
    nome_vendedor VARCHAR(100),
    supervisor_id INT,
    nome_supervisor VARCHAR(100)
);

-- Correção
CREATE TABLE Funcionario (
    id INT PRIMARY KEY,
    nome VARCHAR(100),
    supervisor_id INT,
    FOREIGN KEY (supervisor_id) REFERENCES Funcionario(id)
);

CREATE TABLE Venda (
    id INT PRIMARY KEY,
    vendedor_id INT,
    FOREIGN KEY (vendedor_id) REFERENCES Funcionario(id)
);

```

2. Soluções

- Identificar e eliminar dependências transitivas
- Criar estruturas normalizadas
- Manter documentação clara
- Estabelecer padrões de desenvolvimento

Conclusão

A Terceira Forma Normal é fundamental para:

- Garantir qualidade dos dados
- Facilitar manutenção
- Reduzir redundância
- Melhorar consistência

Deve ser implementada considerando:

- Requisitos do negócio
- Performance do sistema
- Complexidade das consultas
- Necessidades de manutenção

Forma Normal de Boyce-Codd (BCNF)

Definição

A Forma Normal de Boyce-Codd (BCNF) é uma versão mais rigorosa da 3FN que exige:

1. A tabela deve estar na 3FN
2. Toda dependência funcional não-trivial deve ser determinada por uma chave candidata

Conceitos Fundamentais

1. Dependência por Chave Candidata

- Todos os determinantes devem ser chaves candidatas
- Elimina anomalias remanescentes da 3FN
- Garante maior integridade dos dados

2. Determinantes

- Atributos que determinam funcionalmente outros atributos
- Devem ser chaves candidatas
- Base para identificação de violações BCNF

Exemplos Práticos

Exemplo 1: Professor e Disciplina

Violação da BCNF

```
-- Tabela não normalizada
CREATE TABLE Professor_Disciplina (
    professor_id INT,
    disciplina VARCHAR(100),
    departamento VARCHAR(50),
    PRIMARY KEY (professor_id, disciplina)
);

-- Problema: departamento determina disciplina, mas não é chave candidata
```

Aplicação da BCNF

```
-- Tabelas normalizadas
CREATE TABLE Professor_Departamento (
    professor_id INT PRIMARY KEY,
    departamento VARCHAR(50)
);

CREATE TABLE Departamento_Disciplina (
    departamento VARCHAR(50),
    disciplina VARCHAR(100),
    PRIMARY KEY (departamento, disciplina)
);
```

Exemplo 2: Estudante e Curso

Violação da BCNF

```
-- Tabela não normalizada
CREATE TABLE Estudante_Curso (
    estudante_id INT,
    curso_id INT,
    professor VARCHAR(100),
    disciplina VARCHAR(100),
    PRIMARY KEY (estudante_id, curso_id),
```

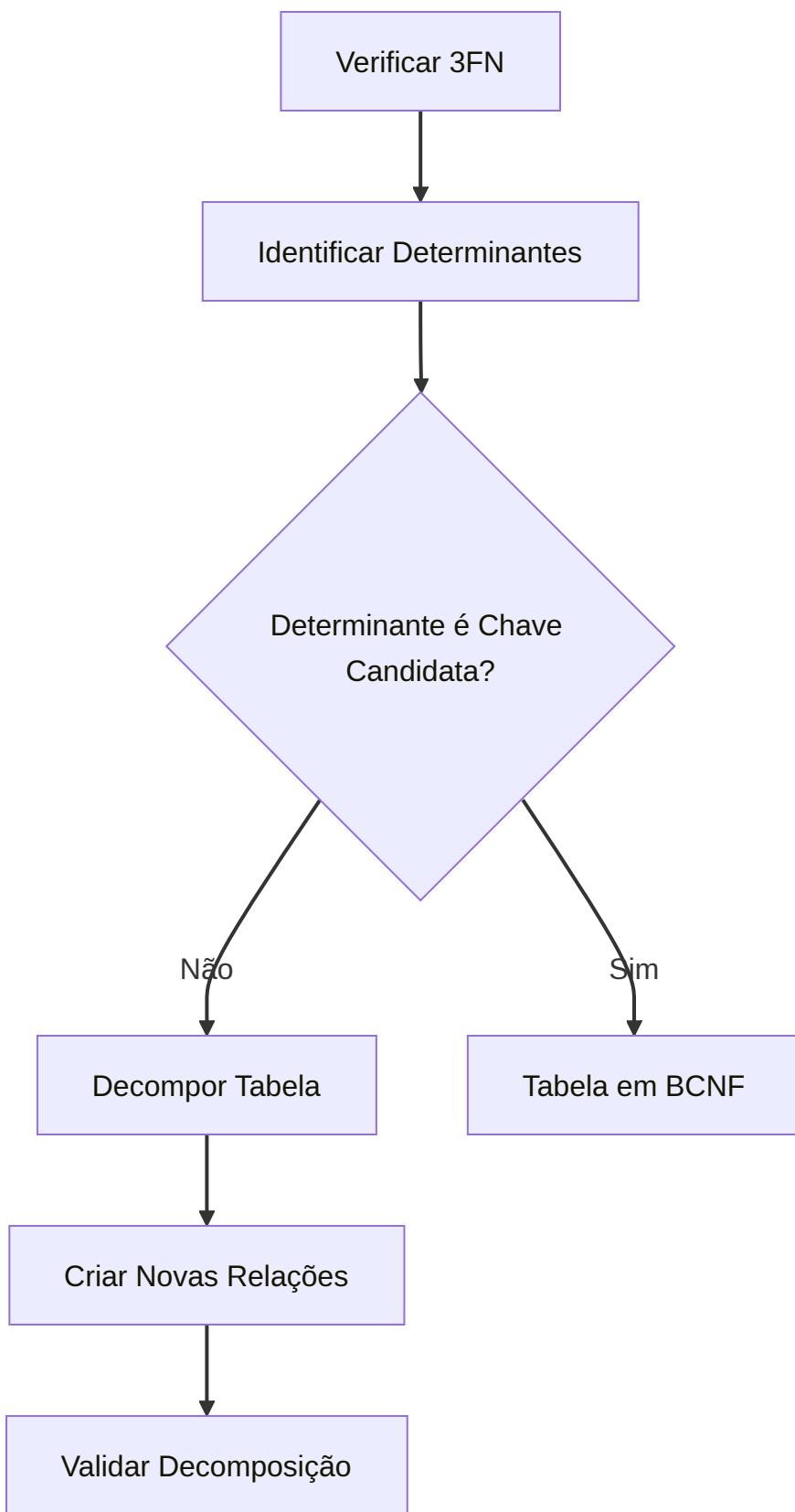
```
-- professor determina disciplina, mas não é chave  
);
```

Aplicação da BCNF

```
-- Tabelas normalizadas  
CREATE TABLE Professor_Disciplina (  
    professor VARCHAR(100) PRIMARY KEY,  
    disciplina VARCHAR(100)  
);  
  
CREATE TABLE Estudante_Curso (  
    estudante_id INT,  
    curso_id INT,  
    professor VARCHAR(100),  
    PRIMARY KEY (estudante_id, curso_id),  
    FOREIGN KEY (professor) REFERENCES  
Professor_Disciplina(professor)  
);
```

Processo de Normalização

1. Identificação de Violações



2. Passos para Normalização

1. Confirmar 3FN
2. Identificar todos os determinantes
3. Verificar se são chaves candidatas
4. Decompor quando necessário
5. Validar preservação de dependências

Benefícios

1. Integridade

- Eliminação de anomalias
- Consistência garantida
- Dependências bem definidas

2. Design

- Estrutura mais limpa
- Relacionamentos claros
- Maior facilidade de manutenção

Considerações Práticas

1. Implementação

```
-- Exemplo de migração para BCNF
INSERT INTO Professor_Departamento (professor_id, departamento)
SELECT DISTINCT professor_id, departamento
FROM professor_disciplina_antiga;

INSERT INTO Departamento_Disciplina (departamento, disciplina)
```

```
SELECT DISTINCT departamento, disciplina  
FROM professor_disciplina_antiga;
```

2. Performance

- Avaliação de joins necessários
- Impacto em consultas complexas
- Balanceamento com requisitos

Checklist de Validação

1. Verificação

- [] Tabela está em 3FN?
- [] Determinantes identificados?
- [] Chaves candidatas definidas?
- [] Decomposição necessária?

2. Testes

- [] Preservação de dados
- [] Integridade mantida
- [] Consultas eficientes
- [] Atualizações consistentes

Anti-Padrões

1. Violações Comuns

```
-- Anti-padrão: Determinante não-chave  
CREATE TABLE Projeto (
```

```

    projeto_id INT,
    gerente_id INT,
    equipe_id INT,
    -- gerente determina equipe, mas não é chave
    PRIMARY KEY (projeto_id)
);

-- Correção
CREATE TABLE Gerente_Equipe (
    gerente_id INT PRIMARY KEY,
    equipe_id INT
);

CREATE TABLE Projeto (
    projeto_id INT PRIMARY KEY,
    gerente_id INT,
    FOREIGN KEY (gerente_id) REFERENCES Gerente_Equipe(gerente_id)
);

```

2. Soluções

- Identificar determinantes não-chave
- Decompor corretamente
- Manter rastreabilidade
- Documentar decisões

Conclusão

A BCNF é importante para:

- Garantir design robusto
- Eliminar anomalias
- Manter integridade

- Facilitar evolução

Deve ser aplicada considerando:

- Complexidade do domínio
- Requisitos de consulta
- Necessidades de performance
- Manutenibilidade

Quarta Forma Normal (4FN)

Definição

A Quarta Forma Normal (4FN) é um nível avançado de normalização que exige:

1. A tabela deve estar na BCNF
2. Não deve haver dependências multivaloradas não-triviais

Conceitos Fundamentais

1. Dependência Multivalorada

- Ocorre quando um atributo determina um conjunto de valores de outro atributo
- Representada como $A \twoheadrightarrow B$ (A determina múltiplos valores de B)
- Independente de outros atributos na relação

2. Independência Mútua

- Atributos multivalorados devem ser independentes entre si
- Não deve haver correlação entre conjuntos de valores
- Base para identificação de violações 4FN

Exemplos Práticos

Exemplo 1: Funcionário e Habilidades

Violação da 4FN

```
-- Tabela não normalizada
CREATE TABLE Funcionario_Habilidade_Projeto (
    funcionario_id INT,
    habilidade VARCHAR(50),
```

```
    projeto VARCHAR(100),
    PRIMARY KEY (funcionario_id, habilidade, projeto)
);
-- Problema: habilidades e projetos são independentes
```

Aplicação da 4FN

```
-- Tabelas normalizadas
CREATE TABLE Funcionario_Habilidade (
    funcionario_id INT,
    habilidade VARCHAR(50),
    PRIMARY KEY (funcionario_id, habilidade)
);

CREATE TABLE Funcionario_Projeto (
    funcionario_id INT,
    projeto VARCHAR(100),
    PRIMARY KEY (funcionario_id, projeto)
);
```

Exemplo 2: Estudante e Atividades

Violação da 4FN

```
-- Tabela não normalizada
CREATE TABLE Estudante_Curso_Atividade (
    estudante_id INT,
    curso VARCHAR(100),
    atividade_extra VARCHAR(100),
    PRIMARY KEY (estudante_id, curso, atividade_extra)
);
```

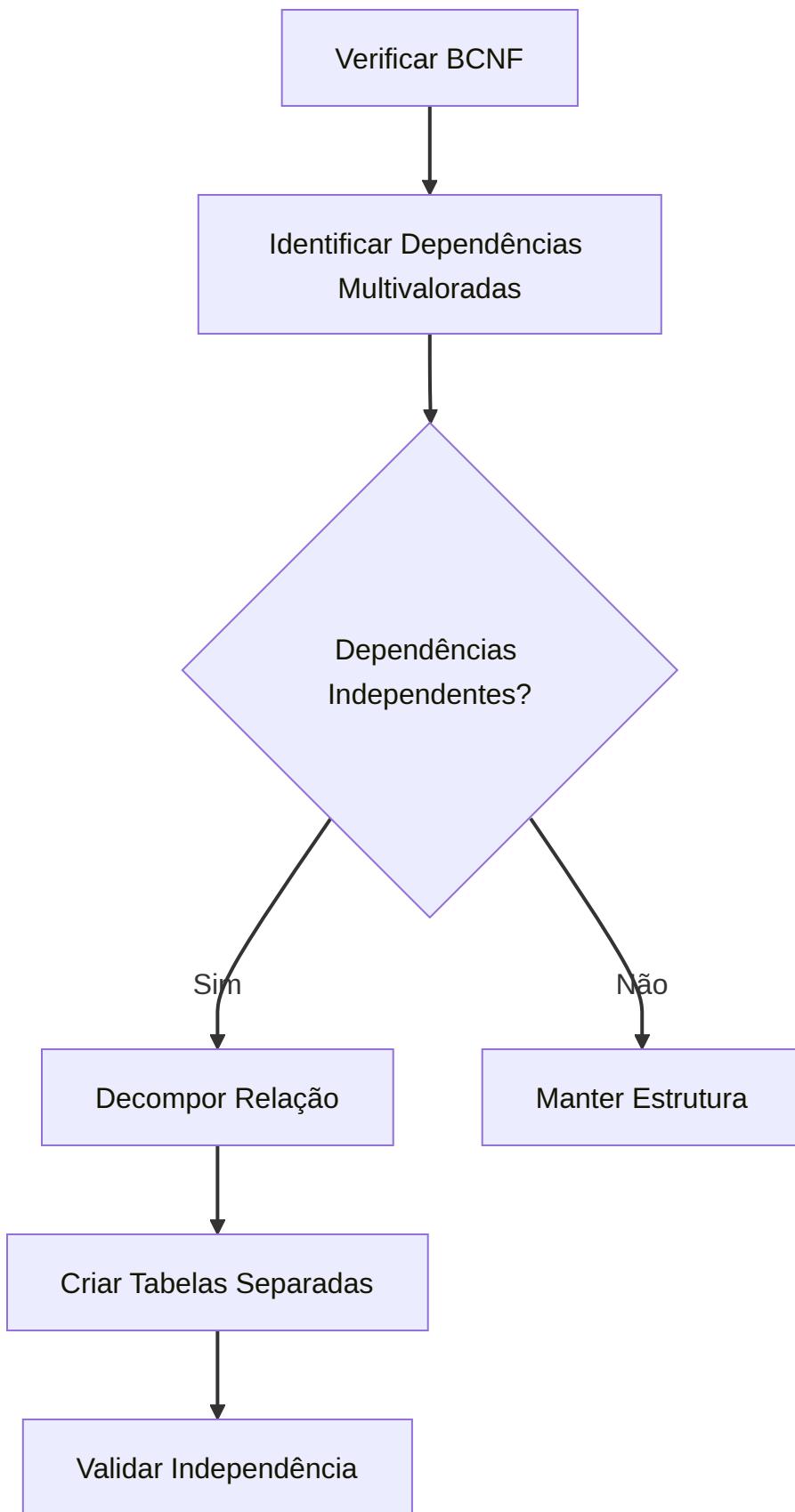
Aplicação da 4FN

```
-- Tabelas normalizadas
CREATE TABLE Estudante_Curso (
    estudante_id INT,
```

```
curso VARCHAR(100),  
PRIMARY KEY (estudante_id, curso)  
);  
  
CREATE TABLE Estudante_Atividade (  
estudante_id INT,  
atividade_extra VARCHAR(100),  
PRIMARY KEY (estudante_id, atividade_extra)  
);
```

Processo de Normalização

1. Identificação de Dependências



2. Passos para Normalização

1. Confirmar BCNF
2. Identificar dependências multivaloradas
3. Verificar independência
4. Decompor quando necessário
5. Validar decomposição

Benefícios

1. Estrutura de Dados

- Eliminação de redundância
- Melhor organização
- Relacionamentos claros

2. Integridade

- Dados consistentes
- Atualizações simplificadas
- Menor risco de anomalias

Considerações Práticas

1. Implementação

```
-- Exemplo de migração para 4FN
INSERT INTO Funcionario_Habilidade (funcionario_id, habilidade)
SELECT DISTINCT funcionario_id, habilidade
FROM funcionario_habilidade_projeto;

INSERT INTO Funcionario_Projeto (funcionario_id, projeto)
```

```
SELECT DISTINCT funcionario_id, projeto  
FROM funcionario_habilidade_projeto;
```

2. Performance

- Análise de impacto
- Necessidade de joins
- Otimização de consultas

Checklist de Validação

1. Verificação

- [] Tabela está em BCNF?
- [] Dependências multivaloradas identificadas?
- [] Independência verificada?
- [] Decomposição adequada?

2. Testes

- [] Integridade mantida
- [] Dados consistentes
- [] Consultas eficientes
- [] Atualizações corretas

Anti-Padrões

1. Violações Comuns

```
-- Anti-padrão: Dependências multivaloradas misturadas  
CREATE TABLE Professor_Disciplina_Livro (
```

```

professor_id INT,
disciplina VARCHAR(100),
livro_referencia VARCHAR(200),
PRIMARY KEY (professor_id, disciplina, livro_referencia)
);

-- Correção
CREATE TABLE Professor_Disciplina (
    professor_id INT,
    disciplina VARCHAR(100),
    PRIMARY KEY (professor_id, disciplina)
);

CREATE TABLE Professor_Livro (
    professor_id INT,
    livro_referencia VARCHAR(200),
    PRIMARY KEY (professor_id, livro_referencia)
);

```

2. Soluções

- Identificar dependências independentes
- Separar em relações distintas
- Manter rastreabilidade
- Documentar decisões

Conclusão

A Quarta Forma Normal é essencial para:

- Eliminar redundâncias complexas
- Garantir independência de dados
- Facilitar manutenção

- Melhorar integridade

Deve ser implementada considerando:

- Complexidade do domínio
- Requisitos de consulta
- Necessidades de performance
- Facilidade de manutenção

Quinta Forma Normal (5FN)

Definição

A Quinta Forma Normal (5FN), também conhecida como Forma Normal de Projeção-Junção (PJNF), é o nível mais alto de normalização que exige:

1. A tabela deve estar na 4FN
2. Não deve haver dependências de junção não-triviais

Conceitos Fundamentais

1. Dependência de Junção

- Ocorre quando uma tabela pode ser reconstruída a partir de suas projeções
- Decomposição sem perda de informação
- Mais complexa que dependências multivaloradas

2. Decomposição por Junção

- Divisão em múltiplas tabelas menores
- Preservação completa da informação
- Reconstrução através de junções naturais

Exemplos Práticos

Exemplo 1: Representante, Fabricante e Produto

Violação da 5FN

```
-- Tabela não normalizada
CREATE TABLE Representante_Fabricante_Produto (
    representante VARCHAR(100),
```

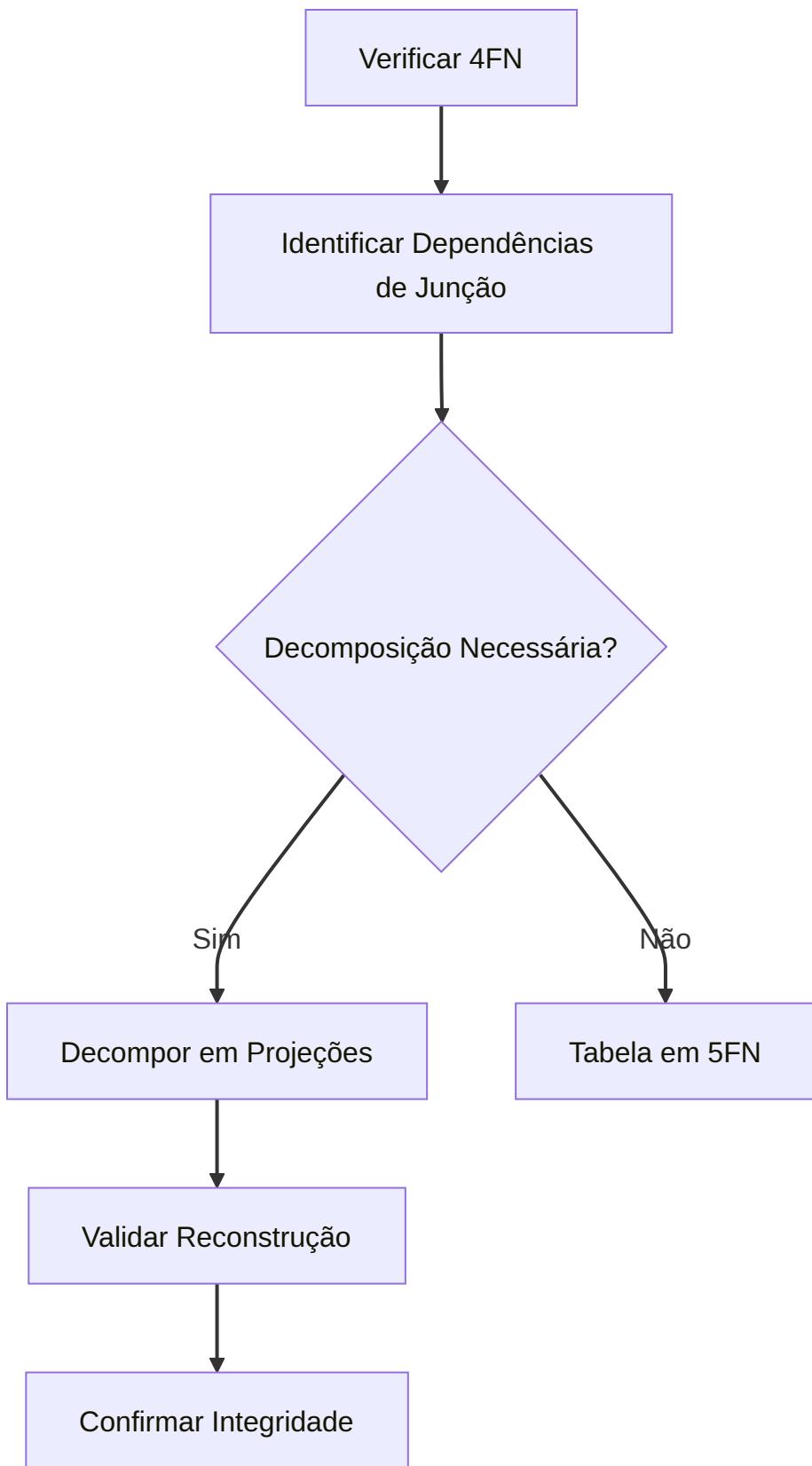
```
fabricante VARCHAR(100),  
produto VARCHAR(100),  
PRIMARY KEY (representante, fabricante, produto)  
);
```

Aplicação da 5FN

```
-- Tabelas normalizadas  
CREATE TABLE Representante_Fabricante (  
    representante VARCHAR(100),  
    fabricante VARCHAR(100),  
    PRIMARY KEY (representante, fabricante)  
);  
  
CREATE TABLE Fabricante_Produto (  
    fabricante VARCHAR(100),  
    produto VARCHAR(100),  
    PRIMARY KEY (fabricante, produto)  
);  
  
CREATE TABLE Representante_Produto (  
    representante VARCHAR(100),  
    produto VARCHAR(100),  
    PRIMARY KEY (representante, produto)  
);
```

Processo de Normalização

1. Identificação de Dependências



2. Passos para Normalização

1. Confirmar 4FN
2. Identificar dependências de junção
3. Avaliar decomposição
4. Criar projeções
5. Validar reconstrução

Benefícios

1. Qualidade dos Dados

- Eliminação total de redundância
- Máxima integridade
- Consistência garantida

2. Design

- Estrutura otimizada
- Relacionamentos puros
- Manutenção simplificada

Considerações Práticas

1. Implementação

```
-- Exemplo de migração para 5FN
INSERT INTO Representante_Fabricante (representante, fabricante)
SELECT DISTINCT representante, fabricante
FROM representante_fabricante_produto;

INSERT INTO Fabricante_Produto (fabricante, produto)
SELECT DISTINCT fabricante, produto
FROM representante_fabricante_produto;
```

```
INSERT INTO Representante_Produto (representante, produto)
SELECT DISTINCT representante, produto
FROM representante_fabricante_produto;
```

2. Desafios

- Complexidade de queries
- Performance de junções
- Manutenção de integridade

Quando Aplicar

1. Cenários Ideais

- Dados altamente inter-relacionados
- Necessidade de máxima integridade
- Atualizações frequentes
- Relacionamentos complexos

2. Considerações

- Custo de implementação
- Impacto na performance
- Complexidade de manutenção
- Necessidades do negócio

Anti-Padrões

1. Violações Comuns

```

-- Anti-padrão: Dependências de junção não decompostas
CREATE TABLE Fornecedor_Peca_Projeto (
    fornecedor_id INT,
    peca_id INT,
    projeto_id INT,
    quantidade INT,
    PRIMARY KEY (fornecedor_id, peca_id, projeto_id)
);

-- Correção
CREATE TABLE Fornecedor_Peca (
    fornecedor_id INT,
    peca_id INT,
    PRIMARY KEY (fornecedor_id, peca_id)
);

CREATE TABLE Peca_Projeto (
    peca_id INT,
    projeto_id INT,
    PRIMARY KEY (peca_id, projeto_id)
);

CREATE TABLE Fornecedor_Projeto (
    fornecedor_id INT,
    projeto_id INT,
    PRIMARY KEY (fornecedor_id, projeto_id)
);

```

2. Soluções

- Análise cuidadosa de dependências
- Decomposição apropriada
- Validação de junções
- Documentação detalhada

Conclusão

1. Importância

- Máximo nível de normalização
- Eliminação total de redundância
- Integridade absoluta dos dados
- Base teórica sólida

2. Aplicação Prática

- Avaliar necessidade real
- Considerar trade-offs
- Balancear com performance
- Documentar decisões

Recomendações Finais

1. Avaliação

- Analisar requisitos do sistema
- Avaliar volume de dados
- Considerar padrões de acesso
- Medir impacto na performance

2. Implementação

- Planejar cuidadosamente
- Testar extensivamente

- Monitorar performance
- Manter documentação

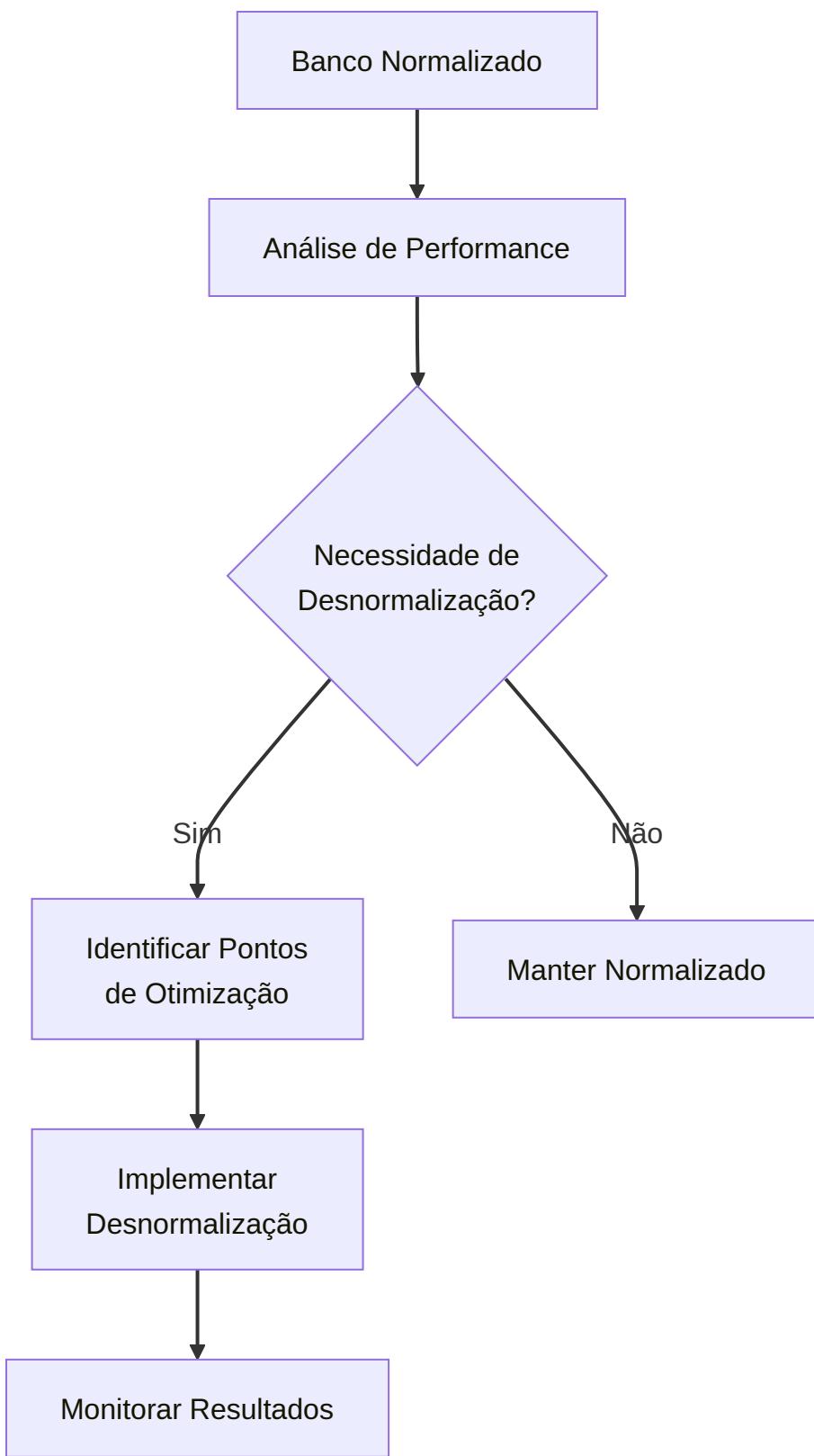
Guia Completo de Desnormalização

Introdução

A desnormalização é uma estratégia deliberada de otimização que introduz redundância controlada em um banco de dados normalizado para melhorar a performance de leitura e simplificar consultas.

Fundamentos

1. Conceitos Básicos



2. Pré-requisitos

- Compreensão das formas normais
- Análise de performance atual
- Identificação de gargalos
- Métricas de baseline

Técnicas de Desnormalização

1. Duplicação de Dados

```
-- Antes (Normalizado)
CREATE TABLE Cliente (
    id INT PRIMARY KEY,
    nome VARCHAR(100),
    endereco VARCHAR(200)
);

CREATE TABLE Pedido (
    id INT PRIMARY KEY,
    cliente_id INT,
    data_pedido DATE,
    FOREIGN KEY (cliente_id) REFERENCES Cliente(id)
);

-- Depois (Desnormalizado)
CREATE TABLE Pedido (
    id INT PRIMARY KEY,
    cliente_id INT,
    cliente_nome VARCHAR(100),
    cliente_endereco VARCHAR(200),
    data_pedido DATE,
    FOREIGN KEY (cliente_id) REFERENCES Cliente(id)
);
```

2. Tabelas Agregadas

```
-- Antes (Normalizado)
CREATE TABLE Venda (
    id INT PRIMARY KEY,
    produto_id INT,
    quantidade INT,
    valor DECIMAL(10, 2),
    data_venda DATE
);

-- Depois (Desnormalizado)
CREATE TABLE Venda_Diaria (
    data DATE PRIMARY KEY,
    total_vendas INT,
    valor_total DECIMAL(10, 2),
    media_valor DECIMAL(10, 2)
);
```

3. Campos Calculados

```
-- Antes (Normalizado)
CREATE TABLE Pedido_Item (
    pedido_id INT,
    produto_id INT,
    quantidade INT,
    valor_unitario DECIMAL(10, 2)
);

-- Depois (Desnormalizado)
CREATE TABLE Pedido_Item (
    pedido_id INT,
    produto_id INT,
    quantidade INT,
    valor_unitario DECIMAL(10, 2),
    valor_total DECIMAL(10, 2),
```

```
percentual_pedido DECIMAL(5,2)  
);
```

Estratégias de Implementação

1. Análise de Requisitos

- Identificar padrões de acesso
- Avaliar frequência de leituras vs escritas
- Mapear consultas críticas
- Definir métricas de sucesso

2. Planejamento



3. Implementação Gradual

1. Começar com mudanças pequenas
2. Testar extensivamente
3. Medir impacto
4. Ajustar conforme necessário

Casos de Uso

1. E-Commerce

```
-- Desnormalização para catálogo de produtos  
CREATE TABLE Produto_Catalogo (  
    id INT PRIMARY KEY,  
    nome VARCHAR(100),  
    preco DECIMAL(10,2),
```

```
        categoria_nome VARCHAR(50),
        marca_nome VARCHAR(50),
        qtd_estoque INT,
        media_avaliacoes DECIMAL(3,2),
        total_vendas INT
);
```

2. Business Intelligence

```
-- Tabela desnormalizada para relatórios
CREATE TABLE Vendas_Analitico (
    data DATE,
    vendedor_nome VARCHAR(100),
    regiao VARCHAR(50),
    produto_categoria VARCHAR(50),
    total_vendas DECIMAL(10,2),
    qtd_itens INT,
    margem_lucro DECIMAL(5,2)
);
```

Manutenção e Monitoramento

1. Sincronização de Dados

```
-- Trigger para manter dados sincronizados
CREATE TRIGGER atualiza_pedido_cliente
AFTER UPDATE ON Cliente
FOR EACH ROW
BEGIN
    UPDATE Pedido
    SET cliente_nome = NEW.nome,
        cliente_endereco = NEW.endereco
    WHERE cliente_id = NEW.id;
END;
```

2. Monitoramento

- Performance de queries
- Uso de espaço em disco
- Consistência de dados
- Tempo de processamento

Boas Práticas

1. Documentação

- Justificativa para desnormalização
- Mapeamento de dependências
- Procedimentos de manutenção
- Impacto nas aplicações

2. Testes

```
-- Exemplo de validação de consistência
CREATE PROCEDURE validar_consistencia()
BEGIN
    SELECT p.cliente_nome, c.nome,
    CASE
        WHEN p.cliente_nome <> c.nome THEN 'Inconsistente'
        ELSE 'OK'
    END as status
    FROM Pedido p
    JOIN Cliente c ON p.cliente_id = c.id;
END;
```

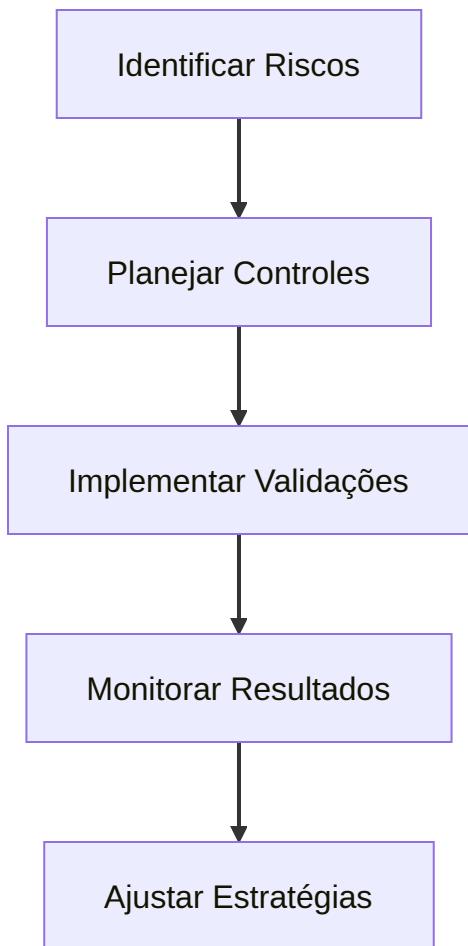
Riscos e Mitigações

1. Riscos Comuns

- Inconsistência de dados

- Aumento do espaço em disco
- Complexidade de manutenção
- Performance de escritas

2. Estratégias de Mitigação



Conclusão

1. Quando Desnormalizar

- Alta carga de leitura
- Relatórios complexos
- Dados históricos
- Performance crítica

2. Quando Evitar

- Dados altamente voláteis
- Consistência crítica
- Recursos limitados
- Manutenção complexa

Checklist de Implementação

1. Preparação

- [] Análise de requisitos completa
- [] Métricas baseline estabelecidas
- [] Plano de implementação definido
- [] Estratégia de rollback preparada

2. Execução

- [] Testes de performance realizados
- [] Procedimentos de sincronização implementados
- [] Documentação atualizada
- [] Monitoramento configurado

3. Pós-Implementação

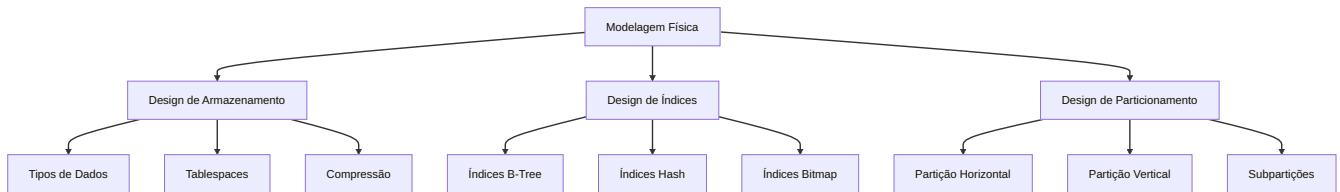
- [] Validação de consistência
- [] Verificação de performance
- [] Treinamento da equipe

- [] Revisão de procedimentos

Modelagem Física

A modelagem física é a última etapa do processo de modelagem de dados, onde o modelo lógico é transformado em uma implementação específica para um SGBD.

Visão Geral



Componentes Principais

1. Design de Armazenamento

- Escolha de tipos de dados
- Configuração de tablespaces
- Estratégias de compressão
- Gestão de espaço

2. Design de Índices

- Seleção de tipos de índices
- Otimização de consultas
- Manutenção de índices
- Monitoramento de uso

3. Design de Particionamento

- Estratégias de particionamento
- Critérios de distribuição

- Gerenciamento de partições
- Otimização de consultas

Considerações de Performance

1. Otimização de I/O

- Distribuição de dados
- Buffer cache
- Prefetch
- Write-back

2. Otimização de CPU

- Processamento paralelo
- Particionamento
- Compressão
- Execução de queries

Melhores Práticas

1. Planejamento de Capacidade

- Crescimento de dados
- Requisitos de performance
- Recursos de hardware
- Janelas de manutenção

2. Monitoramento

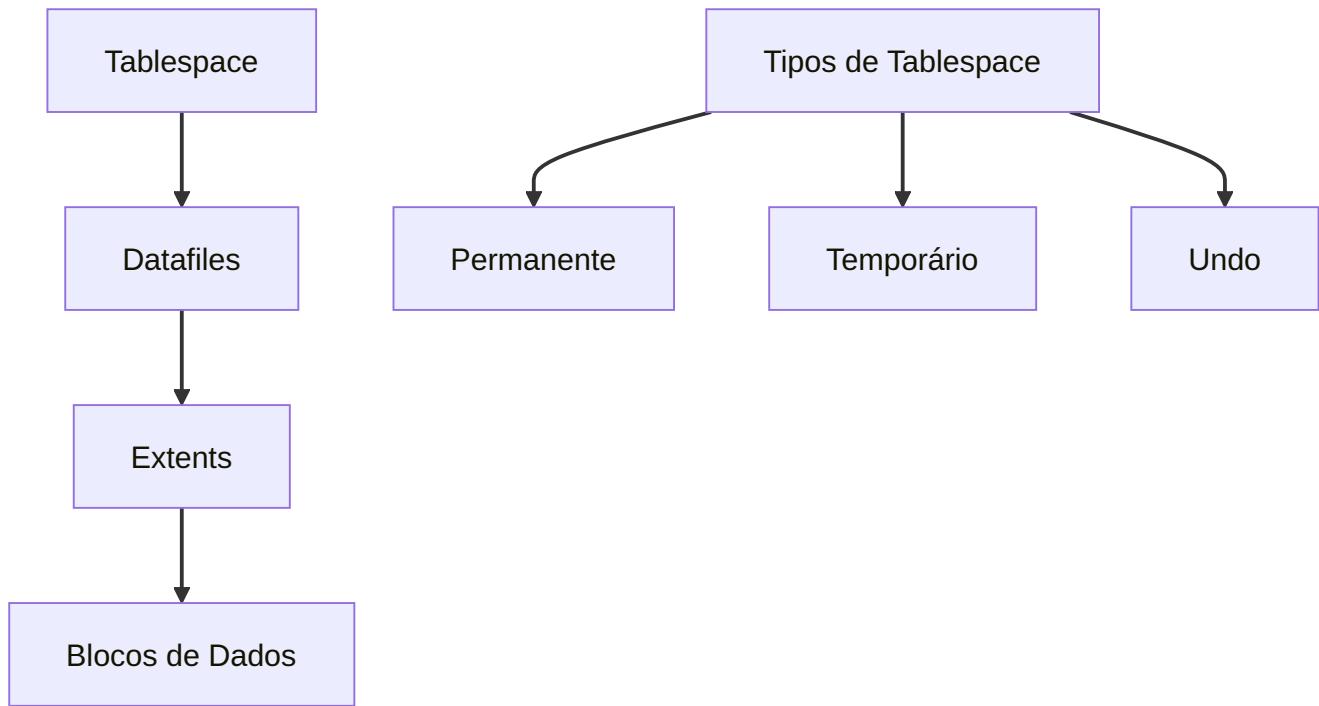
- Uso de espaço

- Performance de queries
- Fragmentação
- Contenção de recursos

Design de Armazenamento

Estruturas de Armazenamento

1. Tablespaces



2. Tipos de Dados

- Numéricos
 - INTEGER, DECIMAL, FLOAT
 - Considerações de precisão
- Caracteres
 - CHAR, VARCHAR, TEXT
 - Codificação e collation
- Data/Hora
 - DATE, TIMESTAMP

- Fusos horários
- Binários
 - BLOB, BINARY
 - Armazenamento externo

3. Compressão de Dados

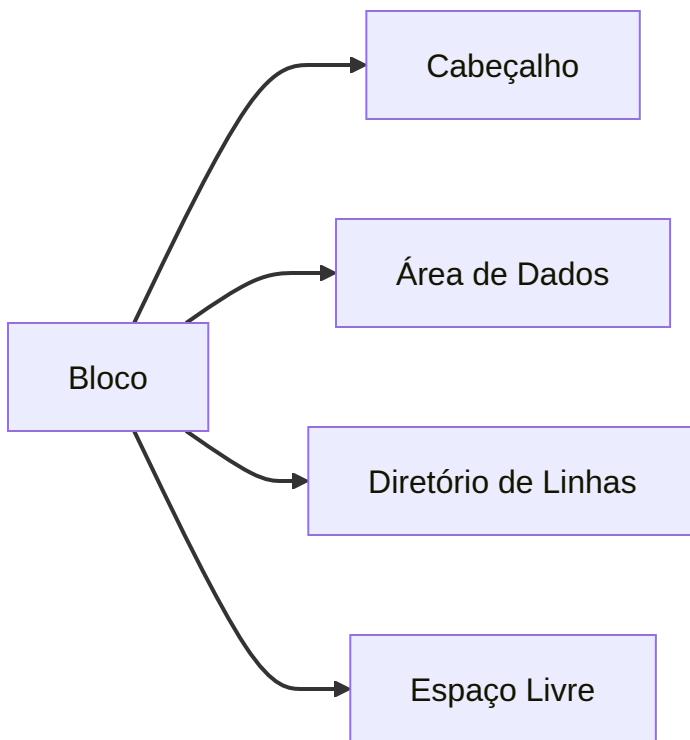
```
-- Exemplo de tabela com compressão
CREATE TABLE vendas_comprimidas (
    id INT,
    data DATE,
    valor DECIMAL(10,2)
) COMPRESS FOR OLTP;
```

Estratégias de Organização

1. Alocação de Espaço

- Initial extent
- Next extent
- PCTFREE
- PCTUSED

2. Gestão de Blocos



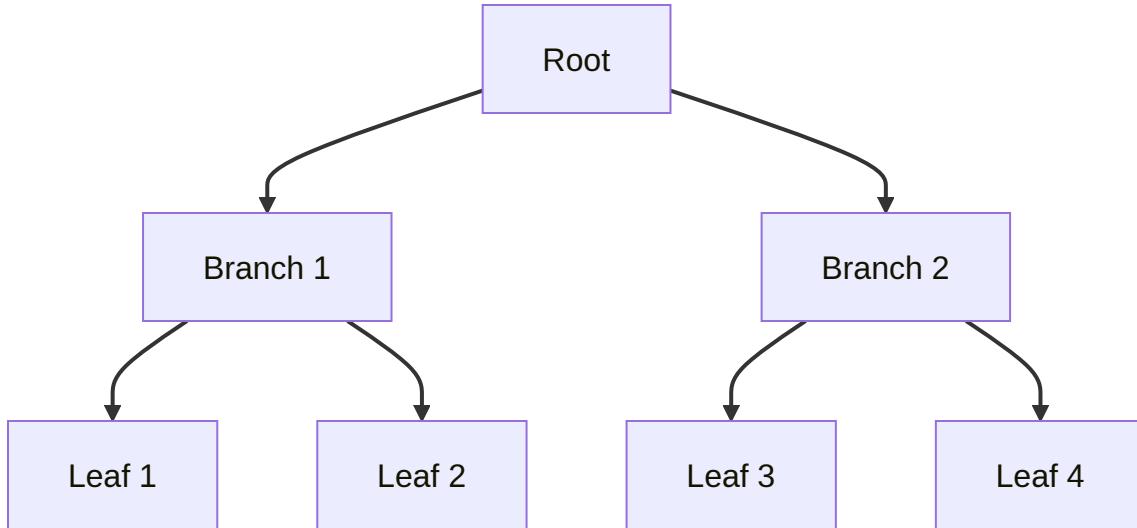
3. Configurações de I/O

- Tamanho de bloco
- Buffer cache
- Direct I/O
- Async I/O

Design de Índices

Tipos de Índices

1. B-Tree



2. Hash

- Tabela de hash
- Função de hash
- Tratamento de colisões
- Casos de uso

3. Bitmap

```
-- Exemplo de índice bitmap
CREATE BITMAP INDEX idx_status
ON pedidos(status)
TABLESPACE index_tbs;
```

Estratégias de Indexação

1. Seleção de Colunas

- Seletividade
- Frequência de acesso
- Padrões de consulta
- Cardinalidade

2. Manutenção

```
-- Reorganização de índice  
ALTER INDEX idx_nome REBUILD;  
  
-- Análise de estatísticas  
ANALYZE TABLE tabela  
COMPUTE STATISTICS FOR ALL INDEXES;
```

3. Monitoramento

- Usage tracking
- Fragmentação
- Hit ratio
- I/O stats

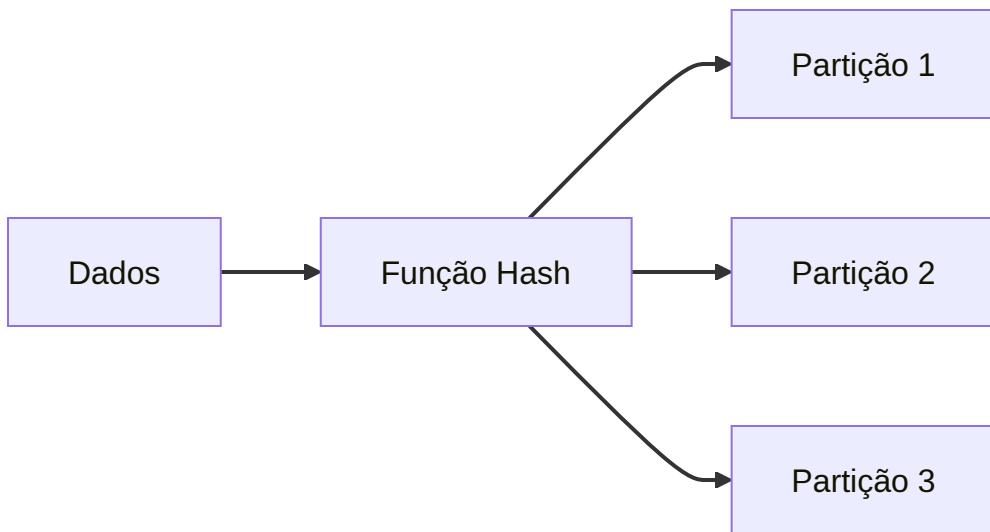
Design de Particionamento

Tipos de Particionamento

1. Particionamento Horizontal

```
-- Exemplo de particionamento por range
CREATE TABLE vendas (
    id INT,
    data DATE,
    valor DECIMAL(10, 2)
) PARTITION BY RANGE (data) (
    PARTITION p2021 VALUES LESS THAN ('2022-01-01'),
    PARTITION p2022 VALUES LESS THAN ('2023-01-01'),
    PARTITION p2023 VALUES LESS THAN ('2024-01-01')
);
```

2. Particionamento por Hash



3. Particionamento Composto

- Range-Hash
- Range-List

- List-Hash

Estratégias de Implementação

1. Critérios de Particionamento

- Data
- ID
- Região
- Status

2. Gerenciamento de Partições

```
-- Adicionar nova partição
ALTER TABLE vendas
ADD PARTITION p2024
VALUES LESS THAN ('2025-01-01');

-- Mesclar partições
ALTER TABLE vendas
MERGE PARTITIONS p2021, p2022
INTO PARTITION p_historico;
```

3. Manutenção

- Rotação de partições
- Arquivamento
- Purge de dados
- Rebalanceamento

Otimização de Consultas

1. Partition Pruning

- Eliminação de partições
- Partition-wise joins
- Parallel processing

2. Monitoramento

```
-- Análise de uso de partições
SELECT partition_name, num_rows, blocks
FROM user_tab_partitions
WHERE table_name = 'VENDAS';
```

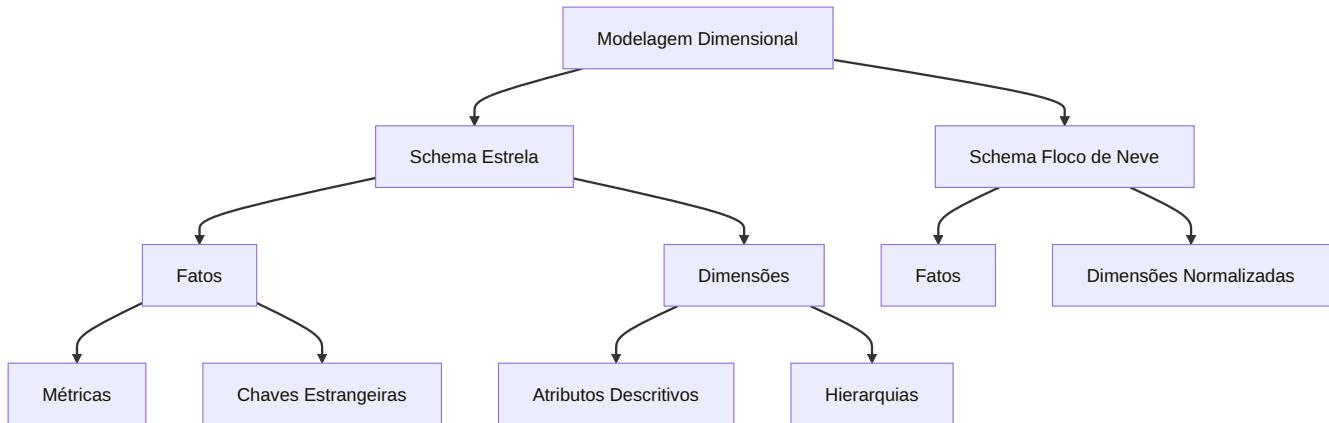
3. Considerações de Performance

- Tamanho das partições
- Distribuição de dados
- I/O balanceamento
- Índices locais vs globais

Modelagem Dimensional

A modelagem dimensional é uma técnica específica para modelagem de data warehouses e data marts, otimizada para consultas analíticas e processamento OLAP.

Visão Geral



Componentes Principais

1. Tabelas Fato

- Métricas de negócio
- Granularidade
- Chaves estrangeiras
- Tipos de fatos

2. Tabelas Dimensão

- Atributos descritivos
- Hierarquias
- Dimensões conformadas
- SCD (Slowly Changing Dimensions)

Tipos de Schema

1. Schema Estrela

- Desnormalizado
- Performance otimizada
- Simplicidade
- Manutenção facilitada

2. Schema Flocos de Neve

- Normalizado
- Economia de espaço
- Complexidade maior
- Mais joins necessários

Melhores Práticas

1. Design de Fatos

- Definir granularidade
- Identificar métricas
- Estabelecer dimensões
- Garantir integridade

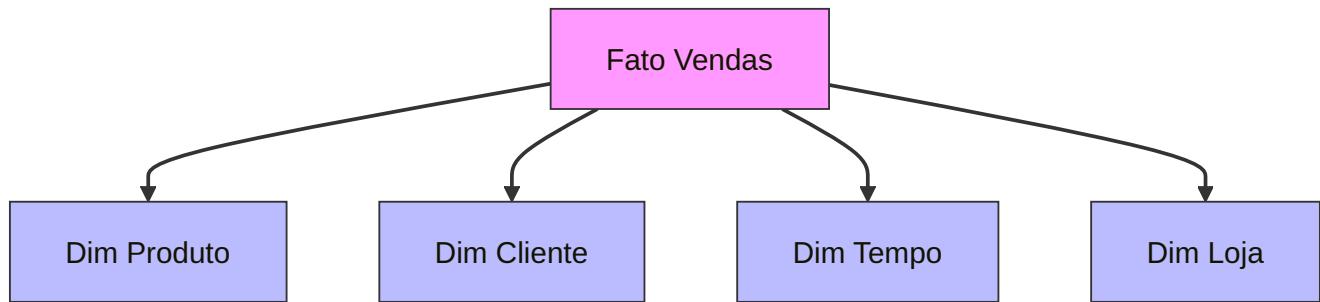
2. Design de Dimensões

- Definir hierarquias
- Planejar mudanças
- Padronizar atributos

- Manter conformidade

Schema Estrela

Estrutura Básica



Características

1. Tabela Fato Central

```
CREATE TABLE fato_vendas (
    sk_produto INT,
    sk_cliente INT,
    sk_tempo INT,
    sk_loja INT,
    quantidade INT,
    valor_venda DECIMAL(10, 2),
    custo DECIMAL(10, 2),
    FOREIGN KEY (sk_produto) REFERENCES dim_produto(sk_produto),
    FOREIGN KEY (sk_cliente) REFERENCES dim_cliente(sk_cliente),
    FOREIGN KEY (sk_tempo) REFERENCES dim_tempo(sk_tempo),
    FOREIGN KEY (sk_loja) REFERENCES dim_loja(sk_loja)
);
```

2. Dimensões Desnormalizadas

- Atributos consolidados
- Hierarquias em uma tabela
- Redundância controlada

- Otimização para queries

Vantagens

1. Performance

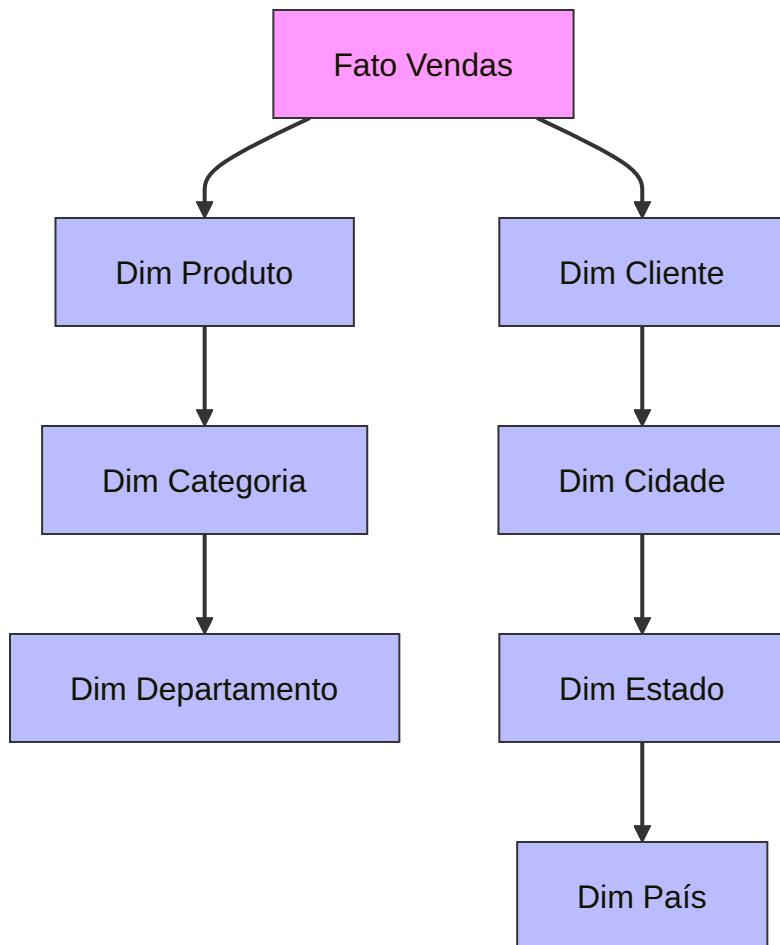
- Menos joins
- Queries simples
- Indexação eficiente
- Cache otimizado

2. Usabilidade

- Fácil entendimento
- Manutenção simples
- Desenvolvimento rápido

Schema Floco de Neve

Estrutura Básica



Características

1. Normalização de Dimensões

```
-- Hierarquia de localização normalizada
CREATE TABLE dim_cidade (
    sk_cidade INT PRIMARY KEY,
    nome_cidade VARCHAR(50),
    sk_estado INT,
    FOREIGN KEY (sk_estado) REFERENCES dim_estado(sk_estado)
);
```

```
CREATE TABLE dim_estado (
    sk_estado INT PRIMARY KEY,
    nome_estado VARCHAR(50),
    sk_pais INT,
    FOREIGN KEY (sk_pais) REFERENCES dim_pais(sk_pais)
);
```

2. Hierarquias Explícitas

- Níveis separados
- Integridade referencial
- Economia de espaço
- Manutenção facilitada

Comparação com Schema Estrela

1. Vantagens

- Menor redundância
- Consistência de dados
- Atualizações eficientes
- Normalização completa

2. Desvantagens

- Performance reduzida
- Mais joins necessários
- Complexidade maior

- Queries mais complexas

Tabelas Fato

Tipos de Fatos

1. Fatos Transacionais

```
CREATE TABLE fato_vendas (
    sk_data INT,
    sk_produto INT,
    sk_cliente INT,
    sk_loja INT,
    quantidade INT,
    valor_venda DECIMAL(10, 2),
    custo DECIMAL(10, 2),
    PRIMARY KEY (sk_data, sk_produto, sk_cliente, sk_loja)
);
```

2. Fatos Periódicos

```
CREATE TABLE fato_estoque_mensal (
    sk_data INT,
    sk_produto INT,
    sk_loja INT,
    quantidade_media INT,
    valor_estoque DECIMAL(10, 2),
    giro_estoque DECIMAL(5, 2)
);
```

3. Fatos Acumulativos

```
CREATE TABLE fato_pedidos (
    sk_pedido INT,
    sk_cliente INT,
    sk_data_pedido INT,
    sk_data_aprovacao INT,
    sk_data_envio INT,
```

```
    sk_data_entrega INT,  
    valor_total DECIMAL(10, 2)  
);
```

Granularidade

1. Níveis Comuns

- Transação individual
- Diário
- Semanal
- Mensal

2. Agregações

```
-- Exemplo de agregação  
CREATE TABLE fato_vendas_diarias AS  
SELECT  
    sk_data,  
    sk_produto,  
    sk_loja,  
    SUM(quantidade) as qtd_total,  
    SUM(valor_venda) as valor_total  
FROM fato_vendas  
GROUP BY sk_data, sk_produto, sk_loja;
```

Métricas

1. Tipos de Métricas

- Aditivas
- Semi-aditivas

- Não-aditivas

2. Cálculos Comuns

```
-- Exemplo de métricas calculadas
SELECT
    d.mes,
    SUM(f.valor_venda) as receita_total,
    AVG(f.valor_venda) as ticket_medio,
    SUM(f.valor_venda - f.custo) as margem_bruta
FROM fato_vendas f
JOIN dim_tempo d ON f.sk_data = d.sk_data
GROUP BY d.mes;
```

Tabelas Dimensão

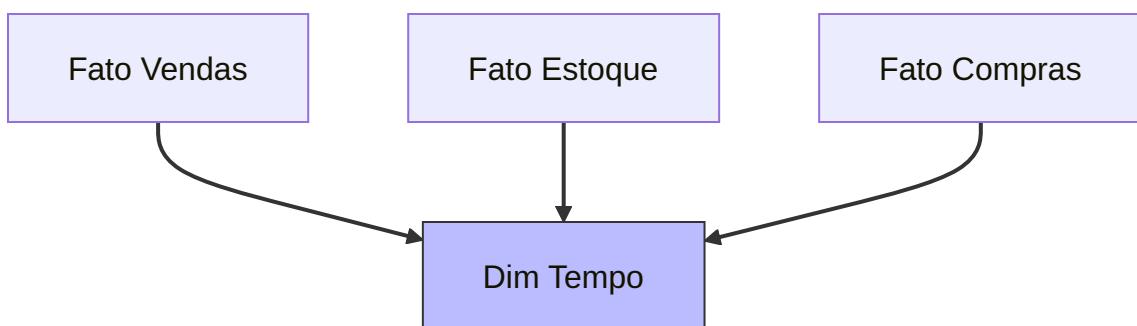
Estrutura Básica

1. Dimensão Produto

```
CREATE TABLE dim_produto (
    sk_produto INT PRIMARY KEY,
    cod_produto VARCHAR(20),
    nome VARCHAR(100),
    marca VARCHAR(50),
    categoria VARCHAR(50),
    subcategoria VARCHAR(50),
    preco_base DECIMAL(10, 2),
    data_inicio DATE,
    data_fim DATE,
    flag_atual CHAR(1)
);
```

Tipos de Dimensões

1. Dimensões Conformadas



2. Role-Playing Dimensions

```
-- Exemplo de views para role-playing
CREATE VIEW dim_data_pedido AS
SELECT * FROM dim_tempo;
```

```
CREATE VIEW dim_data_entrega AS
SELECT * FROM dim_tempo;
```

3. Junk Dimensions

```
CREATE TABLE dim_status (
    sk_status INT PRIMARY KEY,
    status_pedido VARCHAR(20),
    status_pagamento VARCHAR(20),
    tipo_entrega VARCHAR(20),
    prioridade VARCHAR(10)
);
```

Slowly Changing Dimensions (SCD)

1. Tipo 1 (Sobrescrita)

```
UPDATE dim_produto
SET preco_base = 29.99
WHERE sk_produto = 1001;
```

2. Tipo 2 (Histórico)

```
-- Fechando registro atual
UPDATE dim_produto
SET data_fim = CURRENT_DATE,
    flag_atual = 'N'
WHERE sk_produto = 1001
AND flag_atual = 'S';

-- Inserindo novo registro
INSERT INTO dim_produto (
    sk_produto,
    cod_produto,
    nome,
    preco_base,
```

```
    data_inicio,  
    flag_atual  
) VALUES (  
    NEXT_SK(),  
    'PROD1001',  
    'Produto A',  
    29.99,  
    CURRENT_DATE,  
    'S'  
);
```

3. Tipo 3 (Versão Anterior)

```
ALTER TABLE dim_produto  
ADD preco_anterior DECIMAL(10, 2),  
ADD data_alteracao_preco DATE;
```

Hierarquias

1. Definição

```
CREATE TABLE dim_localizacao (   
    sk_localizacao INT PRIMARY KEY,  
    cidade VARCHAR(50),  
    estado VARCHAR(50),  
    regiao VARCHAR(50),  
    pais VARCHAR(50),  
    continente VARCHAR(30)  
);
```

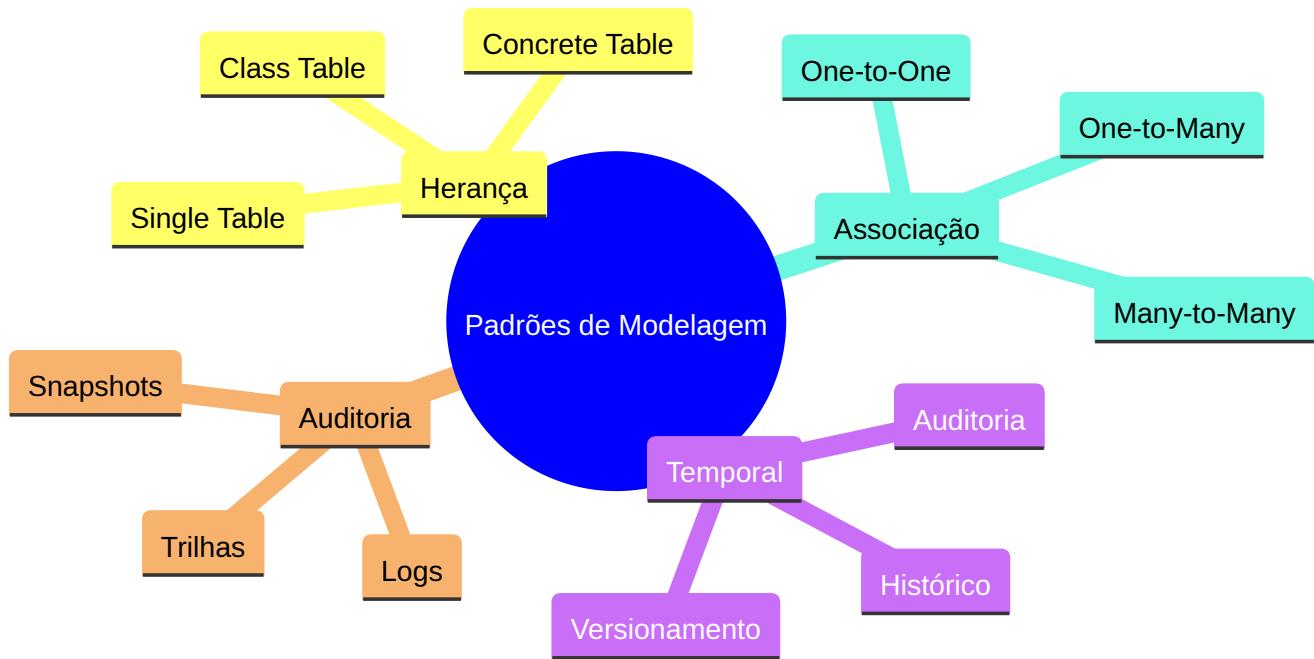
2. Navegação

```
-- Exemplo de drill-down  
SELECT  
    l.continente,  
    l.pais,
```

```
l.regiao,  
    SUM(f.valor_venda) as total_vendas  
FROM fato_vendas f  
JOIN dim_localizacao l ON f.sk_localizacao = l.sk_localizacao  
GROUP BY ROLLUP(l.continente, l.pais, l.regiao);
```

Padrões de Modelagem de Dados

Visão Geral



Categorias de Padrões

1. Padrões de Herança

- Mapeamento de hierarquias
- Polimorfismo em dados
- Reutilização de estruturas

2. Padrões de Associação

- Relacionamentos complexos
- Cardinalidades
- Agregações

3. Padrões Temporais

- Histórico de mudanças
- Versionamento
- Dados temporais

4. Padrões de Auditoria

- Rastreamento
- Segurança
- Conformidade

Seleção de Padrões

Critérios

1. Requisitos Funcionais

- Funcionalidades necessárias
- Regras de negócio
- Casos de uso

2. Requisitos Não-Funcionais

- Performance
- Manutenibilidade
- Escalabilidade

3. Contexto

- Tecnologia
- Equipe

- Restrições

Padrões de Herança

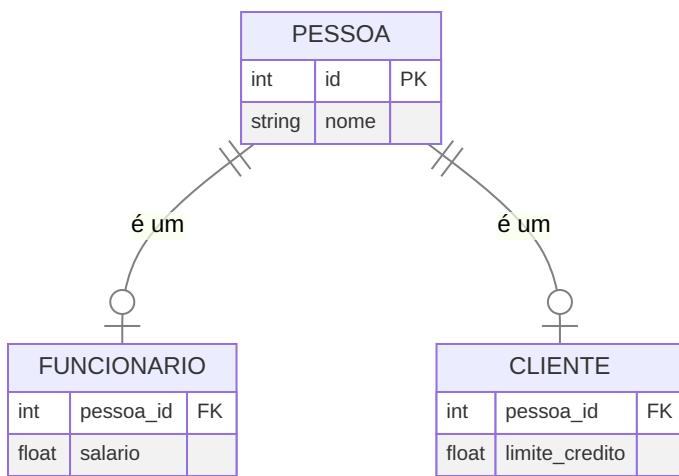
Single Table Inheritance

PESSOA		
int	id	PK
string	nome	
string	tipo	
float	salario	
float	limite_credito	

Implementação

```
CREATE TABLE pessoa (
    id INT PRIMARY KEY,
    nome VARCHAR(100),
    tipo VARCHAR(20),
    salario DECIMAL(10, 2) NULL,
    limite_credito DECIMAL(10, 2) NULL
);
```

Class Table Inheritance



Implementação

```
CREATE TABLE pessoa (
    id INT PRIMARY KEY,
    nome VARCHAR(100)
);

CREATE TABLE funcionario (
    pessoa_id INT PRIMARY KEY,
    salario DECIMAL(10, 2),
    FOREIGN KEY (pessoa_id) REFERENCES pessoa(id)
);

CREATE TABLE cliente (
    pessoa_id INT PRIMARY KEY,
    limite_credito DECIMAL(10, 2),
    FOREIGN KEY (pessoa_id) REFERENCES pessoa(id)
);
```

Concrete Table Inheritance

FUNCIONARIO		
int	id	PK
string	nome	
float	salario	

CLIENTE		
int	id	PK
string	nome	
float	limite_credito	

Implementação

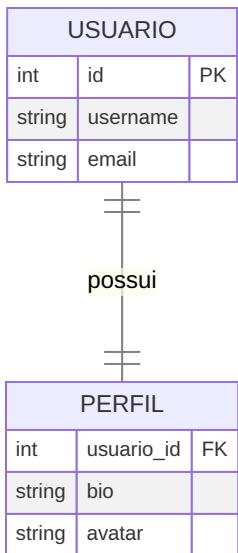
```
CREATE TABLE funcionario (
    id INT PRIMARY KEY,
    nome VARCHAR(100),
    salario DECIMAL(10, 2)
);
```

```
CREATE TABLE cliente (
    id INT PRIMARY KEY,
```

```
    nome VARCHAR(100),  
    limite_credito DECIMAL(10, 2)  
);
```

Padrões de Associação

One-to-One



Implementação

```
CREATE TABLE usuario (
    id INT PRIMARY KEY,
    username VARCHAR(50),
    email VARCHAR(100)
);

CREATE TABLE perfil (
    usuario_id INT PRIMARY KEY,
    bio TEXT,
    avatar VARCHAR(200),
    FOREIGN KEY (usuario_id) REFERENCES usuario(id)
);
```

One-to-Many

Implementação

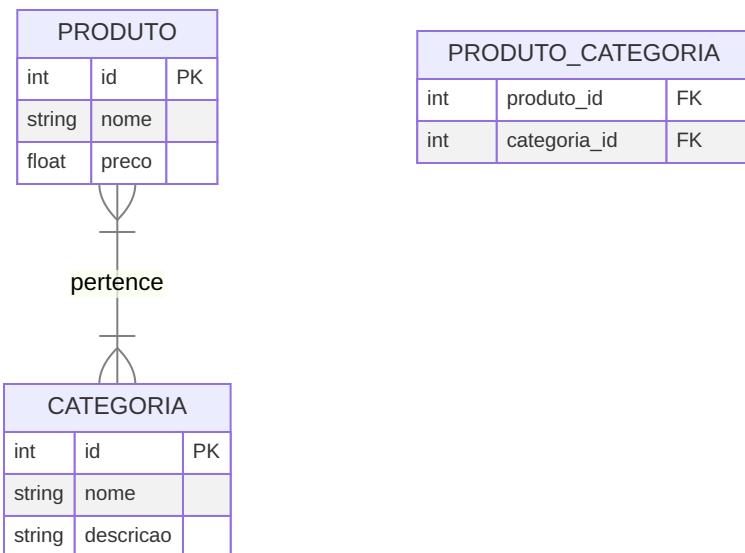
```

CREATE TABLE pedido (
    id INT PRIMARY KEY,
    data DATE,
    total DECIMAL(10,2)
);

CREATE TABLE item (
    id INT PRIMARY KEY,
    pedido_id INT,
    produto VARCHAR(100),
    quantidade INT,
    FOREIGN KEY (pedido_id) REFERENCES pedido(id)
);

```

Many-to-Many



Implementação

```

CREATE TABLE produto (
    id INT PRIMARY KEY,
    nome VARCHAR(100),
    preco DECIMAL(10,2)
);

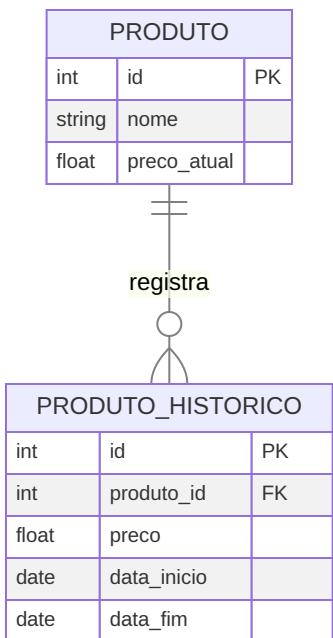
```

```
CREATE TABLE categoria (
    id INT PRIMARY KEY,
    nome VARCHAR(50),
    descricao TEXT
);

CREATE TABLE produto_categoria (
    produto_id INT,
    categoria_id INT,
    PRIMARY KEY (produto_id, categoria_id),
    FOREIGN KEY (produto_id) REFERENCES produto(id),
    FOREIGN KEY (categoria_id) REFERENCES categoria(id)
);
```

Padrões Temporais

Histórico de Mudanças

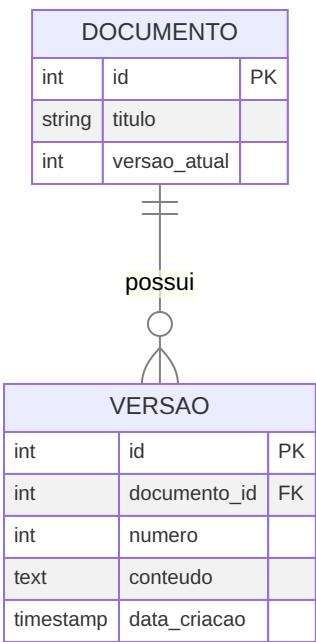


Implementação

```
CREATE TABLE produto (
    id INT PRIMARY KEY,
    nome VARCHAR(100),
    preco_atual DECIMAL(10, 2)
);

CREATE TABLE produto_historico (
    id INT PRIMARY KEY,
    produto_id INT,
    preco DECIMAL(10, 2),
    data_inicio DATE,
    data_fim DATE,
    FOREIGN KEY (produto_id) REFERENCES produto(id)
);
```

Versionamento



Implementação

```
CREATE TABLE documento (
    id INT PRIMARY KEY,
    titulo VARCHAR(200),
    versao_atual INT
);

CREATE TABLE versao (
    id INT PRIMARY KEY,
    documento_id INT,
    numero INT,
    conteudo TEXT,
    data_criacao TIMESTAMP,
    FOREIGN KEY (documento_id) REFERENCES documento(id)
);
```

Dados Temporais

CONTRATO		
int	id	PK
date	data_inicio	
date	data_fim	
string	status	
float	valor	

Implementação

```

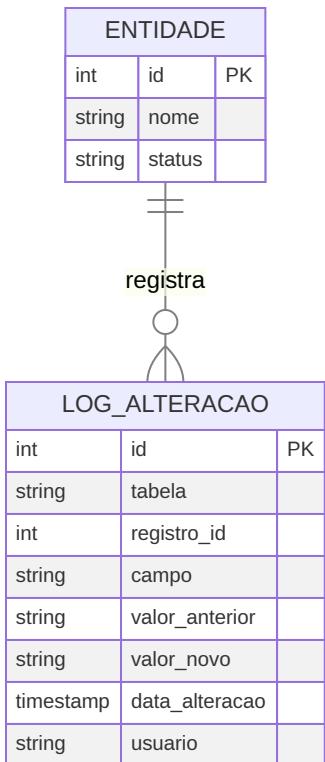
CREATE TABLE contrato (
    id INT PRIMARY KEY,
    data_inicio DATE,
    data_fim DATE,
    status VARCHAR(20),
    valor DECIMAL(10,2)
);

-- Índices para consultas temporais
CREATE INDEX idx_contrato_periodo
ON contrato(data_inicio, data_fim);

```

Padrões de Auditoria

Log de Alterações



Implementação

```
CREATE TABLE log_alteracao (
    id INT PRIMARY KEY,
    tabela VARCHAR(50),
    registro_id INT,
    campo VARCHAR(50),
    valor_anterior TEXT,
    valor_novo TEXT,
    data_alteracao TIMESTAMP,
    usuario VARCHAR(50)
);
```

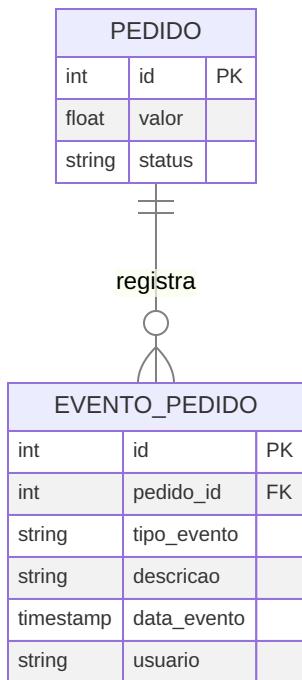
```
-- Trigger de exemplo
```

```

CREATE TRIGGER tr_audit_entidade
AFTER UPDATE ON entidade
FOR EACH ROW
INSERT INTO log_alteracao (
    tabela, registro_id, campo,
    valor_anterior, valor_novo,
    data_alteracao, usuario
) VALUES (
    'entidade', NEW.id, 'status',
    OLD.status, NEW.status,
    CURRENT_TIMESTAMP, CURRENT_USER
);

```

Trilha de Auditoria



Implementação

```

CREATE TABLE evento_pedido (
    id INT PRIMARY KEY,
    pedido_id INT,

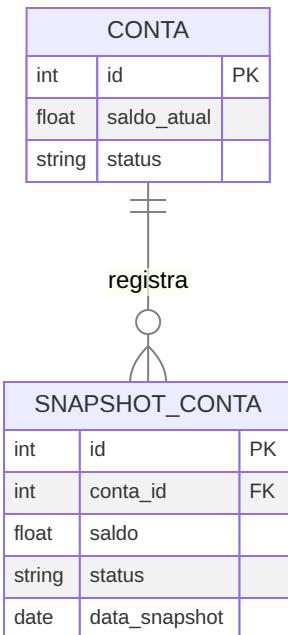
```

```

    tipo_evento VARCHAR(50),
    descricao TEXT,
    data_evento TIMESTAMP,
    usuario VARCHAR(50),
    FOREIGN KEY (pedido_id) REFERENCES pedido(id)
);

```

Snapshot



Implementação

```

CREATE TABLE snapshot_conta (
    id INT PRIMARY KEY,
    conta_id INT,
    saldo DECIMAL(10,2),
    status VARCHAR(20),
    data_snapshot DATE,
    FOREIGN KEY (conta_id) REFERENCES conta(id)
);

```

```
-- Procedure para criar snapshot
```

```
CREATE PROCEDURE criar_snapshot_diario()
BEGIN
    INSERT INTO snapshot_conta (
        conta_id, saldo, status, data_snapshot
    )
    SELECT
        id, saldo_atual, status, CURRENT_DATE
    FROM conta;
END;
```

Fundamentos SQL

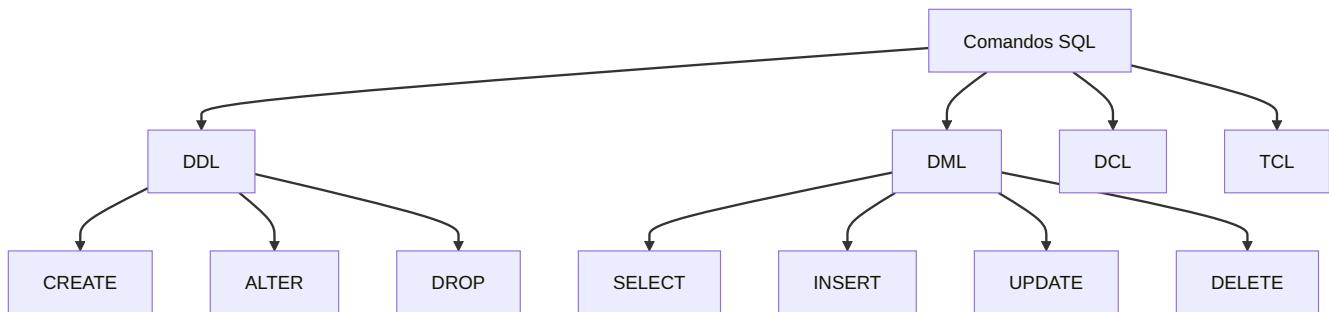
```
|| NEURAL.MATRIX >> SQL.FUNDAMENTOS
|| STATUS: ATIVO
|| SEGURANÇA: CRIPTOGRAFADO
|| ACESSO: CONHECIMENTO_PROFUNDO
||
```

ACID_QUEEN.PERSPECTIVA: Visão Geral

```
CONCEITOS.CORE
```

- ▶ DDL
- ▶ DML
- ▶ DCL
- ▶ TCL

ARQUITETURA.SQL



NOSQL_PUNK.ALERTA: Conceitos Fundamentais

1. DDL (Linguagem de Definição de Dados)

- Manipulação de esquema

- Criação/modificação de objetos
- Controle da estrutura do banco

2. DML (Linguagem de Manipulação de Dados)

- Operações com dados
- Funções CRUD
- Execução de consultas

3. DCL (Linguagem de Controle de Dados)

- Controle de acesso
- Gerenciamento de permissões
- Implementação de segurança

4. TCL (Linguagem de Controle de Transação)

- Gerenciamento de transações
- Consistência de dados
- Propriedades ACID

SEC_PHANTOM.DIRETRIZES: Boas Práticas

PROTÓCOLOS.SEGURANÇA

- ▶ Use prepared statements
- ▶ Valide todas as entradas
- ▶ Configure acessos
- ▶ Monitore performance

TIME_LORD.EXERCÍCIOS: Treinamento Prático

Operações Básicas

```
-- Create table
CREATE TABLE hackers (
    id INT PRIMARY KEY,
    codename VARCHAR(50),
    skill_level INT,
    last_hack TIMESTAMP
);

-- Insert data
INSERT INTO hackers (id, codename, skill_level)
VALUES (1, 'GHOST_PROTOCOL', 9);

-- Query data
SELECT * FROM hackers
WHERE skill_level > 7;

-- Update records
UPDATE hackers
SET last_hack = CURRENT_TIMESTAMP
WHERE id = 1;
```

BACKUP_PRIEST.SABEDORIA: Armadilhas Comuns

Fique Atento A

- Vulnerabilidades de injeção SQL
- Gargalos de performance
- Deadlocks de transação
- Vazamentos de conexão

Soluções

- Use queries parametrizadas
- Implemente indexação adequada
- Monitore timeout de transações
- Implemente pool de conexões

```
|| FIM.DA.TRANSMISSÃO      ||
|| STATUS: COMPLETO        ||
|| PRÓXIMO.MÓDULO: SQL.DDL ||
||
```

DDL - Linguagem de Definição de Dados

A Linguagem de Definição de Dados (DDL) é um componente fundamental do SQL usado para definir e modificar a estrutura do banco de dados. Com ela, podemos criar, alterar e excluir objetos como tabelas, índices, views e outros elementos estruturais.

Principais Comandos DDL

O DDL possui quatro comandos principais que formam a base para gerenciamento de estruturas de banco de dados:

CREATE

O comando CREATE é usado para criar novos objetos no banco de dados. É o comando mais básico e essencial do DDL.

Exemplos práticos:

```
-- Criação de banco de dados
CREATE DATABASE loja;

-- Criação de tabela com diferentes tipos de dados e constraints
CREATE TABLE produtos (
    id INT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    preco DECIMAL(10,2),
    categoria_id INT,
    FOREIGN KEY (categoria_id) REFERENCES categorias(id)
);
```

ALTER

O ALTER permite modificar a estrutura de objetos existentes. É crucial para evolução do schema do banco de dados.

Exemplos comuns:

```
-- Adicionando uma nova coluna  
ALTER TABLE produtos  
ADD COLUMN descricao TEXT;  
  
-- Modificando o tipo de uma coluna  
ALTER TABLE produtos  
ALTER COLUMN preco TYPE NUMERIC(12,2);
```

DROP

Usado para remover objetos do banco de dados. Deve ser usado com extrema cautela pois é irreversível.

Exemplo de uso:

```
-- Removendo uma tabela  
DROP TABLE IF EXISTS produtos_antigos;
```

TRUNCATE

Remove todos os registros de uma tabela, mas mantém sua estrutura. É mais eficiente que DELETE para limpeza completa.

```
TRUNCATE TABLE logs_temporarios;
```

Boas Práticas de DDL

1. Segurança em Primeiro Lugar

- Sempre faça backup antes de operações DDL
- Teste comandos em ambiente de desenvolvimento
- Mantenha scripts de rollback preparados

2. Planejamento

- Documente todas as alterações estruturais

- Avalie impactos em aplicações existentes
- Considere volumes de dados nas operações

3. Versionamento

- Mantenha controle de versão dos schemas
- Use migrations para alterações estruturais
- Documente a evolução do banco de dados

Considerações de Performance

Impacto das Operações

- Operações DDL podem bloquear tabelas
- Alterações em tabelas grandes requerem planejamento
- Considere janelas de manutenção para operações críticas

Otimizações

- Use índices adequadamente
- Planeje particionamento quando necessário
- Considere clustering quando apropriado

Exercício Prático

Vamos criar uma estrutura básica para um sistema de vendas:

```
-- Schema do sistema
CREATE SCHEMA vendas;

-- Tabela de produtos
CREATE TABLE vendas.produtos (
    id SERIAL PRIMARY KEY,
```

```
nome VARCHAR(100) NOT NULL,  
preco DECIMAL(10,2) CHECK (preco > 0),  
estoque INT DEFAULT 0  
);  
  
-- Adicionando índice para busca por nome  
CREATE INDEX idx_produtos_nome  
ON vendas.produtos(nome);
```

Conclusão

O DDL é fundamental para o gerenciamento de bancos de dados, permitindo criar e manter estruturas de dados de forma eficiente. O domínio desses comandos, junto com boas práticas de uso, é essencial para qualquer desenvolvedor ou DBA.



Dica de Segurança: Sempre mantenha backups atualizados e teste suas operações DDL em ambiente de desenvolvimento antes de aplicar em produção.

CREATE Statements: Construindo Estruturas de Dados

```
|| NEURAL.MATRIX >> SQL.CREATE  
|| INSTRUTOR: DATA_ARCHITECT_SAGE
```

Introdução ao CREATE

O comando CREATE é a base para construção de estruturas em bancos de dados. Ele permite criar diversos objetos como databases, tabelas, índices, views e outros elementos essenciais.

CREATE DATABASE

O DATA_ARCHITECT_SAGE explica: "Começamos sempre pelo início - a criação do banco de dados. É como construir a fundação de uma casa."

```
CREATE DATABASE ecommerce  
WITH  
ENCODING = 'UTF8'  
LC_COLLATE = 'pt_BR.UTF-8'  
LC_CTYPE = 'pt_BR.UTF-8'  
TEMPLATE = template0;
```

Por que estes parâmetros?

- `ENCODING`: Garante suporte a caracteres especiais
- `LC_COLLATE` e `LC_CTYPE`: Configurações para português brasileiro
- `TEMPLATE`: Base limpa para novo banco

CREATE TABLE

SCHEMA_MASTER diz: "As tabelas são o coração do seu banco de dados. Vamos criar uma estrutura robusta para um e-commerce."

Exemplo Básico

```
CREATE TABLE produtos (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    preco DECIMAL(10,2) NOT NULL CHECK (preco >= 0),
    descricao TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Exemplo Avançado com Relacionamentos

```
CREATE TABLE pedidos (
    id SERIAL PRIMARY KEY,
    cliente_id INTEGER NOT NULL,
    status VARCHAR(20) NOT NULL,
    valor_total DECIMAL(10,2) NOT NULL,
    data_pedido TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    CONSTRAINT fk_cliente
        FOREIGN KEY (cliente_id)
        REFERENCES clientes(id)
        ON DELETE RESTRICT,

    CONSTRAINT chk_status
        CHECK (status IN ('pendente', 'aprovado', 'cancelado')),

    CONSTRAINT chk_valor
        CHECK (valor_total > 0)
);
```

SECURITY_GUARDIAN acrescenta: "Observe as constraints - são fundamentais para integridade dos dados!"

CREATE INDEX

INDEX_MASTER compartilha: "Índices são como o índice de um livro - essenciais para encontrar informações rapidamente."

Tipos de Índices

-- Índice básico

```
CREATE INDEX idx_produtos_nome ON produtos(nome);
```

-- Índice único

```
CREATE UNIQUE INDEX idx_usuarios_email ON usuarios(email);
```

-- Índice composto

```
CREATE INDEX idx_pedidos_cliente_data ON pedidos(cliente_id, data_pedido);
```

-- Índice parcial

```
CREATE INDEX idx_produtos_ativos ON produtos(nome) WHERE status = 'ativo';
```

CREATE VIEW

QUERY_MASTER explica: "Views são consultas pré-definidas que simplificam o acesso aos dados."

```
CREATE VIEW vw_pedidos_detalhados AS
SELECT
    p.id as pedido_id,
    c.nome as cliente_nome,
    p.valor_total,
    p.status,
    COUNT(i.id) as total_itens
FROM pedidos p
JOIN clientes c ON p.cliente_id = c.id
```

```
JOIN itens_pedido i ON p.id = i_pedido_id  
GROUP BY p.id, c.nome, p.valor_total, p.status;
```

CREATE SEQUENCE

DB_WIZARD diz: "Sequences são úteis para gerar números únicos de forma automática."

```
CREATE SEQUENCE seq_codigo_produto  
INCREMENT BY 1  
START WITH 1000  
NO MINVALUE  
NO MAXVALUE  
CACHE 1;
```

Exercícios Práticos

TRAINING_MASTER propõe: "Vamos praticar com um cenário real!"

Sistema de Blog

```
-- Criando tabela de autores  
CREATE TABLE autores (  
    id SERIAL PRIMARY KEY,  
    nome VARCHAR(100) NOT NULL,  
    email VARCHAR(255) UNIQUE NOT NULL,  
    bio TEXT,  
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
);  
  
-- Criando tabela de posts  
CREATE TABLE posts (  
    id SERIAL PRIMARY KEY,  
    titulo VARCHAR(200) NOT NULL,  
    conteudo TEXT NOT NULL,  
    autor_id INTEGER NOT NULL,  
    status VARCHAR(20) DEFAULT 'rascunho',  
    publicado_em TIMESTAMP,
```

```

CONSTRAINT fk_autor
    FOREIGN KEY (autor_id)
    REFERENCES autores(id),

CONSTRAINT chk_status
    CHECK (status IN ('rascunho', 'publicado', 'arquivado'))
);

-- Criando índices estratégicos
CREATE INDEX idx_posts_autor ON posts(autor_id);
CREATE INDEX idx_posts_status ON posts(status);
CREATE INDEX idx_posts_publicacao ON posts(publicado_em DESC);

-- Criando view para posts publicados
CREATE VIEW vw_posts_publicados AS
SELECT
    p.id,
    p.titulo,
    a.nome as autor_nome,
    p.publicado_em
FROM posts p
JOIN autores a ON p.autor_id = a.id
WHERE p.status = 'publicado'
ORDER BY p.publicado_em DESC;

```

Boas Práticas

BEST_PRACTICES_GURU compartilha dicas essenciais:

1. Nomenclatura

- Use nomes descritivos
- Siga um padrão consistente
- Prefira minúsculas para objetos

- Use underscores para separar palavras

2. Constraints

- Nomeie todas as constraints importantes
- Use CHECK para validações de domínio
- Implemente FKs com cuidado
- Considere ON DELETE/UPDATE actions

3. Índices

- Crie índices com propósito
- Evite índices redundantes
- Nomeie índices de forma descritiva
- Considere o impacto na escrita

LEMBRE-SE:

- Teste antes em desenvolvimento
- Mantenha documentação
- Faça backup antes de criar
- Planeje crescimento

ALTER Statements: Modificando Estruturas de Dados

SCHEMA_EVOLUTION_MASTER >> Modificações Estruturais
INSTRUTOR: DATABASE_ARCHITECT

Fundamentos do ALTER

DATABASE_ARCHITECT explica: "O comando ALTER é sua ferramenta para evolução do banco de dados. Com ele, você pode modificar estruturas existentes sem perder dados."

ALTER TABLE

SCHEMA_MASTER apresenta: "Vamos explorar as principais operações de alteração de tabelas."

1. Adicionando Colunas

```
-- Adicionando uma coluna simples
ALTER TABLE produtos
ADD COLUMN descricao TEXT;

-- Adicionando múltiplas colunas
ALTER TABLE usuarios
ADD COLUMN ultimo_acesso TIMESTAMP,
ADD COLUMN tentativas_login INTEGER DEFAULT 0,
ADD COLUMN bloqueado BOOLEAN DEFAULT FALSE;

-- Adicionando coluna com constraint
ALTER TABLE pedidos
ADD COLUMN valor_total DECIMAL(10,2) NOT NULL DEFAULT 0.0
CHECK (valor_total >= 0);
```

2. Modificando Colunas

MIGRATION_SPECIALIST adverte: "Cuidado ao modificar tipos de dados - certifique-se da compatibilidade!"

```
-- Alterando tipo de dados
ALTER TABLE produtos
ALTER COLUMN preco TYPE NUMERIC(12,2);

-- Modificando valor default
ALTER TABLE usuarios
ALTER COLUMN status SET DEFAULT 'ativo';

-- Removendo valor default
ALTER TABLE logs
ALTER COLUMN nivel DROP DEFAULT;

-- Tornando coluna NOT NULL
ALTER TABLE clientes
ALTER COLUMN email SET NOT NULL;
```

3. Constraints

INTEGRITY GUARDIAN compartilha: "Constraints garantem a qualidade dos seus dados."

```
-- Adicionando Primary Key
ALTER TABLE produtos
ADD CONSTRAINT pk_produtos PRIMARY KEY (id);

-- Adicionando Foreign Key
ALTER TABLE pedidos
ADD CONSTRAINT fk_cliente
FOREIGN KEY (cliente_id)
REFERENCES clientes(id)
ON DELETE RESTRICT;

-- Adicionando Unique Constraint
```

```
ALTER TABLE usuarios
ADD CONSTRAINT uq_email UNIQUE (email);

-- Adicionando Check Constraint
ALTER TABLE produtos
ADD CONSTRAINT chk_preco
CHECK (preco_venda > preco_custo);
```

4. Renomeando Objetos

REFACTORING_MASTER diz: "Às vezes precisamos reorganizar nossa estrutura."

```
-- Renomeando tabela
ALTER TABLE usuarios
RENAME TO users;

-- Renomeando coluna
ALTER TABLE produtos
RENAME COLUMN descricao TO detalhes;

-- Renomeando constraint
ALTER TABLE pedidos
RENAME CONSTRAINT fk_cliente TO fk_pedidos_cliente;
```

ALTER INDEX

INDEX_WIZARD explica: "Índices também precisam de manutenção."

```
-- Renomeando índice
ALTER INDEX idx_old_name
RENAME TO idx_new_name;

-- Modificando configurações do índice
ALTER INDEX idx_produtos
SET (fillfactor = 90);
```

ALTER SEQUENCE

SEQUENCE_MASTER compartilha: "Ajuste suas sequences conforme necessário."

```
-- Modificando sequence
ALTER SEQUENCE seq_pedidos
INCREMENT BY 10
MAXVALUE 999999
CYCLE;
```

Cenários Práticos

PRACTICAL_GURU apresenta: "Vamos ver alguns cenários comuns do mundo real!"

Evolução de Sistema de E-commerce

```
-- Adicionando suporte a múltiplos endereços
ALTER TABLE clientes
ADD COLUMN endereco_entrega JSONB,
ADD COLUMN endereco_cobranca JSONB;

-- Implementando soft delete
ALTER TABLE produtos
ADD COLUMN deleted_at TIMESTAMP,
ADD COLUMN active BOOLEAN DEFAULT TRUE;

-- Adicionando auditoria
ALTER TABLE pedidos
ADD COLUMN created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
ADD COLUMN updated_at TIMESTAMP,
ADD COLUMN created_by INTEGER,
ADD COLUMN updated_by INTEGER;

-- Adicionando constraints de auditoria
ALTER TABLE pedidos
ADD CONSTRAINT fk_created_by
    FOREIGN KEY (created_by) REFERENCES usuarios(id),
```

```
ADD CONSTRAINT fk_updated_by  
FOREIGN KEY (updated_by) REFERENCES usuarios(id);
```

Boas Práticas

BEST_PRACTICES_SAGE compartilha diretrizes essenciais:

1. Segurança

- Faça backup antes de alterações
- Teste em ambiente de desenvolvimento
- Planeje janelas de manutenção
- Prepare scripts de rollback

2. Performance

- Considere o impacto em tabelas grandes
- Avalie bloqueios necessários
- Use transações apropriadamente
- Monitore uso de recursos

3. Manutenção

- Documente todas as alterações
- Mantenha scripts de migração
- Use controle de versão
- Comunique mudanças à equipe

|| CHECKLIST DE ALTERAÇÕES: ||

- || Backup realizado? ||
- || Testado em desenvolvimento? ||
- || Script de rollback preparado? ||
- || Impacto analisado? ||
- || Equipe notificada? ||
- || Janela de manutenção agendada? ||

Troubleshooting Comum

ERROR_HANDLER apresenta soluções para problemas frequentes:

1. Erro de Dependência

```
-- Verificando dependências
SELECT * FROM pg_depend
WHERE objid = 'sua_tabela'::regclass;

-- Removendo dependências com cautela
DROP VIEW IF EXISTS view_dependente CASCADE;
```

2. Problemas de Bloqueio

```
-- Verificando bloqueios
SELECT * FROM pg_locks
WHERE relation = 'sua_tabela'::regclass;

-- Finalizando sessões bloqueantes (com cautela!)
SELECT pg_terminate_backend(pid);
```

Conclusão

DATABASE_ARCHITECT conclui: "O ALTER é poderoso, mas requer responsabilidade. Sempre planeje suas alterações e siga as boas práticas para manter seu banco de dados

saudável e evolutivo."



Dica Final: Mantenha um histórico de todas as alterações estruturais em seu banco de dados. Isso será valioso para troubleshooting futuro e para entender a evolução do sistema.

DROP Statements: Removendo Objetos do Banco de Dados

|| CLEANUP_MASTER >> Remoção de Objetos
|| INSTRUTOR: DATABASE_CLEANER

Fundamentos do DROP

DATABASE_CLEANER alerta: "O comando DROP é poderoso e irreversível. Use com extrema cautela!"

DROP TABLE

SCHEMA_CLEANER apresenta: "Vamos explorar como remover tabelas de forma segura."

```
-- Remoção simples
DROP TABLE IF EXISTS produtos;

-- Remoção com CASCADE
DROP TABLE clientes CASCADE;

-- Remoção múltipla
DROP TABLE IF EXISTS
    temp_logs,
    old_backups,
    test_data;
```

DROP DATABASE

DATABASE_MASTER adverte: "Este é o comando mais perigoso - use com extrema cautela!"

```
-- Removendo banco de dados
DROP DATABASE IF EXISTS test_db;

-- Forçando desconexão de usuários
SELECT pg_terminate_backend(pid)
FROM pg_stat_activity
WHERE datname = 'test_db';
```

DROP INDEX

INDEX_MASTER explica: "Índices desnecessários podem prejudicar a performance."

```
-- Removendo índice
DROP INDEX IF EXISTS idx_produtos_nome;

-- Removendo múltiplos índices
DROP INDEX idx_temp1, idx_temp2;

-- Removendo índice concorrentemente
DROP INDEX CONCURRENTLY idx_logs;
```

DROP CONSTRAINT

CONSTRAINT GUARDIAN orienta: "Remova constraints com cuidado para manter a integridade."

```
-- Removendo constraint
ALTER TABLE pedidos
DROP CONSTRAINT fk_cliente;

-- Removendo PRIMARY KEY
ALTER TABLE produtos
DROP CONSTRAINT produtos_pkey;

-- Removendo UNIQUE constraint
```

```
ALTER TABLE usuarios
DROP CONSTRAINT uq_email;
```

Cenários Práticos

PRACTICAL_MASTER apresenta: "Vamos ver situações reais de limpeza de banco."

Limpeza de Ambiente de Teste

```
-- Removendo objetos temporários
DROP TABLE IF EXISTS temp_import;
DROP INDEX IF EXISTS idx_temp_search;
DROP VIEW IF EXISTS vw_test_report;

-- Removendo schema de teste
DROP SCHEMA test_env CASCADE;
```

Refatoração de Sistema

```
-- Removendo estruturas antigas
DROP TABLE IF EXISTS legacy_customers CASCADE;
DROP VIEW IF EXISTS old_reports;
DROP TRIGGER audit_trigger ON transactions;
```

Boas Práticas

SAFETY GUARDIAN compartilha diretrizes essenciais:

1. Segurança

- Sempre faça backup antes
- Use IF EXISTS para evitar erros
- Verifique dependências
- Documente todas as remoções

2. Performance

- Considere o momento adequado
- Avalie impacto em outros objetos
- Use CONCURRENTLY quando possível
- Monitore locks e conexões

3. Manutenção

- Mantenha scripts de recriação
- Atualize documentação
- Comunique alterações
- Mantenha histórico de mudanças

CHECKLIST DE REMOÇÃO:

- Backup realizado?
- Dependências verificadas?
- Scripts de recriação prontos?
- Equipe notificada?
- Janela de manutenção definida?
- Impacto analisado?

Troubleshooting

ERROR_HANDLER apresenta soluções para problemas comuns:

1. Objetos com Dependências

```
-- Verificando dependências
```

```
SELECT DISTINCT dependent_ns.nspname as dependent_schema,  
dependent_view.relname as dependent_view
```

```
FROM pg_depend
JOIN pg_class dependent_view ON dependent_view.oid =
pg_depend.objid
JOIN pg_namespace dependent_ns ON dependent_ns.oid =
dependent_view.relnamespace
WHERE refobjid = 'sua_tabela'::regclass;
```

2. Objetos Bloqueados

```
-- Identificando bloqueios
SELECT blocked_locks.pid AS blocked_pid,
       blocking_locks.pid AS blocking_pid
FROM pg_catalog.pg_locks blocked_locks
JOIN pg_catalog.pg_locks blocking_locks ON blocking_locks.locktype =
blocked_locks.locktype
WHERE NOT blocked_locks.granted ;
```

Conclusão

DATABASE_CLEANER conclui: "O DROP é uma ferramenta poderosa para manutenção do banco de dados, mas deve ser usado com responsabilidade e planejamento adequado."

⚠ Dica Final: Sempre mantenha scripts de recriação para objetos importantes que precisam ser removidos. Em caso de necessidade, você poderá restaurá-los facilmente.

TRUNCATE: Limpeza Rápida de Dados

|| DATA_CLEANER >> Limpeza Eficiente de Dados
|| INSTRUTOR: TRUNCATE_SPECIALIST

Fundamentos do TRUNCATE

TRUNCATE_SPECIALIST explica: "TRUNCATE é a forma mais rápida de remover todos os dados de uma tabela!"

```
-- Sintaxe básica
TRUNCATE TABLE logs;

-- Múltiplas tabelas
TRUNCATE TABLE
    temp_data,
    staging_area,
    import_buffer;

-- Com RESTART IDENTITY
TRUNCATE TABLE produtos RESTART IDENTITY;

-- Com CASCADE
TRUNCATE TABLE clientes CASCADE;
```

TRUNCATE vs DELETE

PERFORMANCE_GURU compara: "Entenda quando usar cada um!"

TRUNCATE	DELETE
Mais rápido	Mais flexível
Não é transacional*	Transacional
Reseta sequências	Mantém sequências
Não dispara triggers	Dispara triggers

Cenários de Uso

DATA_ARCHITECT apresenta: "Situações ideais para TRUNCATE"

1. Limpeza de Ambiente

```
-- Limpando tabelas temporárias
TRUNCATE TABLE
    temp_imports,
    staging_area,
    etl_buffer
RESTART IDENTITY;
```

2. Reset de Desenvolvimento

```
-- Resetando ambiente de testes
TRUNCATE TABLE
    test_data,
    test_logs,
    test_metrics
CASCADE;
```

Boas Práticas

SAFETY_MASTER compartilha diretrizes essenciais:

1. Segurança

- Faça backup antes
- Verifique dependências
- Use em ambiente correto
- Documente operações

2. Performance

- Considere locks
- Avalie impacto
- Planeje janela de execução
- Monitore espaço em disco

CHECKLIST DE TRUNCATE:

- Backup realizado?
- Ambiente correto?
- Dependências verificadas?
- Equipe notificada?
- Janela de manutenção definida?

Troubleshooting

ERROR_HANDLER apresenta soluções comuns:

1. Bloqueios

```
-- Verificando locks
SELECT relation::regclass, mode, granted
FROM pg_locks
WHERE relation IN (
```

```
SELECT oid
  FROM pg_class
 WHERE relname = 'sua_tabela'
);
```

2. Dependências

```
-- Verificando referências
SELECT
    tc.table_schema,
    tc.table_name,
    kcu.column_name
  FROM information_schema.table_constraints tc
  JOIN information_schema.key_column_usage kcu
    ON tc.constraint_name = kcu.constraint_name
  WHERE tc.constraint_type = 'FOREIGN KEY'
  AND kcu.referenced_table_name = 'sua_tabela';
```

Padrões de Uso

PATTERN_MASTER apresenta: "Padrões comuns de utilização"

1. Limpeza Periódica

```
-- Procedure de limpeza
CREATE PROCEDURE limpar_logs()
LANGUAGE SQL
AS $$

    TRUNCATE TABLE
        system_logs,
        audit_logs,
        error_logs;

$$;
```

2. Reset de Ambiente

```
-- Script de reset
BEGIN;
    -- Desativa foreign key checks
    SET CONSTRAINTS ALL DEFERRED;

    -- Limpa todas as tabelas
    TRUNCATE TABLE
        tabela1,
        tabela2,
        tabela3
    CASCADE;

    -- Reativa constraints
    SET CONSTRAINTS ALL IMMEDIATE;
COMMIT;
```

Conclusão

TRUNCATE_SPECIALIST conclui: "TRUNCATE é uma ferramenta poderosa para limpeza de dados, mas deve ser usada com conhecimento e cautela."



Dica Final: Sempre tenha um plano de recuperação antes de executar TRUNCATE em tabelas importantes. A operação é rápida, mas irreversível!

Exercícios de DDL (Data Definition Language)

Visão Geral

Este módulo contém uma série progressiva de exercícios para praticar comandos DDL em SQL. Os exercícios estão organizados em três níveis de dificuldade:

Nível Básico

- Criação de tabelas simples
- Alterações básicas de estrutura
- Operações fundamentais de DDL

Nível Intermediário

- Relacionamentos entre tabelas
- Constraints complexas
- Modificações de schema

Nível Avançado

- Otimização de estruturas
- Migrations complexas
- Cenários empresariais

Estrutura dos Exercícios

Cada exercício segue o formato:

1. Descrição do problema

2. Requisitos específicos
3. Dicas de implementação
4. Solução de referência
5. Critérios de avaliação

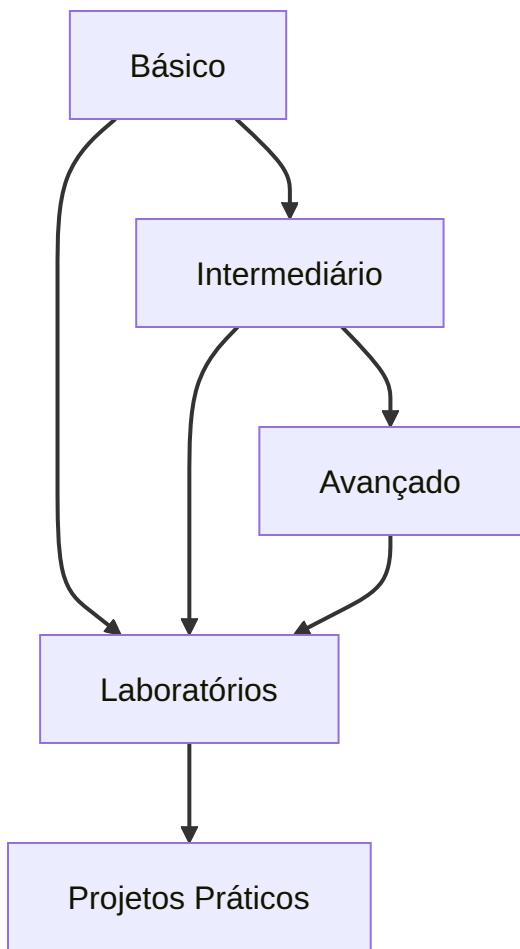
Ambiente de Prática

```
-- Database de teste
CREATE DATABASE exercicios_ddl;

-- Schema para isolamento
CREATE SCHEMA pratica;

-- Tabela de controle
CREATE TABLE controle_exercicios (
    id SERIAL PRIMARY KEY,
    exercicio VARCHAR(50),
    completado BOOLEAN DEFAULT FALSE,
    data_conclusao TIMESTAMP
);
```

Fluxo de Estudo Recomendado



Avaliação de Progresso

Nível	Pontos por Exercício	Total de Exercícios	Pontuação Máxima
Básico	10	10	100
Intermediário	20	10	200
Avançado	30	10	300

Recursos Necessários

1. Ambiente de Desenvolvimento

- PostgreSQL 12+
- Cliente SQL
- Editor de texto

2. Conhecimentos Prévios

- Conceitos básicos de SQL
- Modelagem de dados
- Comandos DDL fundamentais

Navegação do Conteúdo

- Exercícios Básicos ([Exercícios Básicos de DDL](#))
- Exercícios Intermediários ([Exercícios Intermediários de DDL](#))
- Exercícios Avançados ([Exercícios Avançados de DDL](#))

Dicas de Estudo

BOAS PRÁTICAS:

- ✓ Teste cada comando individualmente
- ✓ Mantenha scripts de rollback
- ✓ Documente suas soluções
- ✓ Pratique regularmente
- ✓ Revise conceitos quando necessário

Supporte e Recursos

- Fórum de discussão
- Documentação oficial

- Exemplos práticos
- Soluções comentadas

Próximos Passos

1. Comece pelos exercícios básicos
2. Avance gradualmente
3. Pratique em laboratórios
4. Desenvolva projetos reais

Certificação

Complete todos os exercícios para receber:

- Certificado de conclusão
- Badge de proficiência
- Pontuação para ranking

Feedback

Sua opinião é importante:

- Avalie os exercícios
- Sugira melhorias
- Reporte problemas
- Compartilhe experiências

Exercícios Básicos de DDL

Exercício 1: Criação de Tabela Simples

Descrição

Crie uma tabela para armazenar informações básicas de produtos.

Requisitos

- Nome do produto (máximo 100 caracteres, obrigatório)
- Preço (decimal com 2 casas decimais, obrigatório)
- Descrição (texto livre, opcional)
- Data de cadastro (data/hora automática)

Solução

```
CREATE TABLE produtos (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    preco DECIMAL(10,2) NOT NULL,
    descricao TEXT,
    data_cadastro TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Exercício 2: Alteração de Estrutura

Descrição

Modifique a tabela de produtos adicionando novas colunas.

Requisitos

- Adicionar coluna para quantidade em estoque

- Adicionar coluna para código de barras (único)
- Adicionar coluna para status do produto

Solução

```
ALTER TABLE produtos
    ADD COLUMN estoque INTEGER DEFAULT 0,
    ADD COLUMN codigo_barras VARCHAR(13) UNIQUE,
    ADD COLUMN status VARCHAR(20) DEFAULT 'ativo';
```

Exercício 3: Constraints Básicas

Descrição

Adicione restrições básicas à tabela de produtos.

Requisitos

- Preço não pode ser negativo
- Estoque não pode ser negativo
- Status deve ser 'ativo' ou 'inativo'

Solução

```
ALTER TABLE produtos
    ADD CONSTRAINT check_preco CHECK (preco >= 0),
    ADD CONSTRAINT check_estoque CHECK (estoque >= 0),
    ADD CONSTRAINT check_status CHECK (status IN ('ativo',
    'inativo'));
```

Exercício 4: Tabelas Relacionadas

Descrição

Crie uma tabela de categorias e relate com os produtos.

Requisitos

- Tabela de categorias com nome e descrição
- Relacionamento entre produtos e categorias
- Categoria é obrigatória para produtos

Solução

```
CREATE TABLE categorias (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(50) NOT NULL UNIQUE,
    descricao TEXT
);

ALTER TABLE produtos
    ADD COLUMN categoria_id INTEGER NOT NULL,
    ADD CONSTRAINT fk_categoria
        FOREIGN KEY (categoria_id)
        REFERENCES categorias(id);
```

Exercício 5: Índices Simples

Descrição

Crie índices para melhorar a performance das consultas.

Requisitos

- Índice para busca por nome do produto
- Índice para código de barras
- Índice para categoria

Solução

```
CREATE INDEX idx_produtos_nome ON produtos(nome);
CREATE INDEX idx_produtos_codigo ON produtos(codigo_barras);
CREATE INDEX idx_produtos_categoria ON produtos(categoria_id);
```

Exercício 6: Exclusão e Recriação

Descrição

Pratique operações de remoção e recriação de objetos.

Requisitos

- Remover índices criados
- Remover constraints
- Remover tabelas com dependências

Solução

```
DROP INDEX IF EXISTS idx_produtos_nome;
DROP INDEX IF EXISTS idx_produtos_codigo;
DROP INDEX IF EXISTS idx_produtos_categoria;

ALTER TABLE produtos
    DROP CONSTRAINT IF EXISTS fk_categoria,
    DROP CONSTRAINT IF EXISTS check_preco,
    DROP CONSTRAINT IF EXISTS check_estoque,
    DROP CONSTRAINT IF EXISTS check_status;

DROP TABLE IF EXISTS produtos CASCADE;
DROP TABLE IF EXISTS categorias CASCADE;
```

Exercício 7: Schema e Database

Descrição

Crie um novo schema e database para isolamento.

Requisitos

- Novo database para testes
- Schema específico para produtos
- Mover objetos entre schemas

Solução

```
CREATE DATABASE loja_teste;  
  
\c loja_teste  
  
CREATE SCHEMA produtos_schema;  
  
CREATE TABLE produtos_schema.produtos (  
    id SERIAL PRIMARY KEY,  
    nome VARCHAR(100) NOT NULL  
);
```

Exercício 8: Sequences

Descrição

Trabalhe com sequences personalizadas.

Requisitos

- Sequence para código do produto
- Sequence com valor inicial específico
- Alteração de sequence existente

Solução

```
CREATE SEQUENCE seq_codigo_produto  
    START WITH 1000
```

```
INCREMENT BY 1
NO MINVALUE
NO MAXVALUE
CACHE 1;

ALTER TABLE produtos
    ALTER COLUMN id SET DEFAULT nextval('seq_codigo_produto');
```

Exercício 9: Comentários e Documentação

Descrição

Adicione comentários aos objetos do banco.

Requisitos

- Comentários em tabelas
- Comentários em colunas
- Comentários em constraints

Solução

```
COMMENT ON TABLE produtos IS 'Tabela de cadastro de produtos';
COMMENT ON COLUMN produtos.nome IS 'Nome do produto';
COMMENT ON CONSTRAINT check_preco ON produtos IS 'Garante preço
não negativo';
```

Exercício 10: Visão Consolidada

Descrição

Crie uma estrutura completa integrando todos os conceitos.

Requisitos

- Tabelas relacionadas

- Constraints apropriadas
- Índices necessários
- Documentação completa

Solução

```
-- Schema dedicado
CREATE SCHEMA IF NOT EXISTS loja;

-- Tabela de categorias
CREATE TABLE loja.categorias (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(50) NOT NULL UNIQUE,
    descricao TEXT,
    ativa BOOLEAN DEFAULT true,
    data_criacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Tabela de produtos
CREATE TABLE loja.produtos (
    id SERIAL PRIMARY KEY,
    codigo_barras VARCHAR(13) UNIQUE,
    nome VARCHAR(100) NOT NULL,
    descricao TEXT,
    preco DECIMAL(10,2) NOT NULL CHECK (preco >= 0),
    estoque INTEGER DEFAULT 0 CHECK (estoque >= 0),
    categoria_id INTEGER NOT NULL REFERENCES loja.categorias(id),
    status VARCHAR(20) DEFAULT 'ativo' CHECK (status IN ('ativo',
    'inativo')),
    data_cadastro TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Índices
CREATE INDEX idx_produtos_nome ON loja.produtos(nome);
CREATE INDEX idx_produtos_categoria ON
loja.produtos(categoria_id);
```

```
-- Comentários  
COMMENT ON SCHEMA loja IS 'Schema para sistema de loja';  
COMMENT ON TABLE loja.produtos IS 'Cadastro de produtos da loja';  
COMMENT ON TABLE loja.categorias IS 'Categorias de produtos';
```

Critérios de Avaliação

- Sintaxe SQL correta
- Uso apropriado de constraints
- Nomenclatura adequada
- Documentação clara
- Scripts de rollback

Dicas de Estudo

- Teste cada comando separadamente
- Verifique a estrutura criada
- Documente as decisões tomadas
- Pratique os scripts de rollback

Exercícios Intermediários de DDL

Exercício 1: Herança de Tabelas

Descrição

Implemente uma estrutura de herança para diferentes tipos de produtos.

Requisitos

- Tabela base de produtos
- Tabelas específicas para produtos físicos e digitais
- Constraints específicas para cada tipo

Solução

```
CREATE TABLE produtos_base (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    preco DECIMAL(10,2) NOT NULL,
    data_cadastro TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE produtos_fisicos (
    peso DECIMAL(10,2) NOT NULL,
    dimensoes VARCHAR(50),
    localizacao_estoque VARCHAR(50)
) INHERITS (produtos_base);

CREATE TABLE produtos_digitais (
    tamanho_arquivo VARCHAR(20),
    formato VARCHAR(20),
    url_download VARCHAR(255)
) INHERITS (produtos_base);
```

Exercício 2: Particionamento de Tabelas

Descrição

Crie uma tabela particionada para histórico de vendas.

Requisitos

- Particionamento por intervalo de datas
- Partições mensais
- Índices apropriados para cada partição

Solução

```
CREATE TABLE historico_vendas (
    id SERIAL,
    data_venda DATE NOT NULL,
    produto_id INTEGER,
    quantidade INTEGER,
    valor_total DECIMAL(10,2)
) PARTITION BY RANGE (data_venda);

CREATE TABLE vendas_202301 PARTITION OF historico_vendas
    FOR VALUES FROM ('2023-01-01') TO ('2023-02-01');

CREATE TABLE vendas_202302 PARTITION OF historico_vendas
    FOR VALUES FROM ('2023-02-01') TO ('2023-03-01');

CREATE INDEX idx_vendas_202301_data ON vendas_202301(data_venda);
CREATE INDEX idx_vendas_202302_data ON vendas_202302(data_venda);
```

Exercício 3: Domínios Personalizados

Descrição

Defina domínios personalizados para tipos de dados comuns.

Requisitos

- Domínio para e-mail
- Domínio para telefone
- Domínio para CPF/CNPJ

Solução

```
CREATE DOMAIN email_type AS VARCHAR(255)
    CHECK (VALUE ~ '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$');

CREATE DOMAIN telefone_type AS VARCHAR(20)
    CHECK (VALUE ~ '^\\+[0-9]{10,14}$');

CREATE DOMAIN documento_type AS VARCHAR(14)
    CHECK (VALUE ~ '^\\d{11}$' OR VALUE ~ '^\\d{14}$');

CREATE TABLE clientes (
    id SERIAL PRIMARY KEY,
    email email_type NOT NULL,
    telefone telefone_type,
    documento documento_type NOT NULL UNIQUE
);
```

Exercício 4: Triggers e Funções

Descrição

Implemente triggers para auditoria de alterações.

Requisitos

- Tabela de auditoria
- Trigger para INSERT/UPDATE/DELETE

- Registro de usuário e timestamp

Solução

```

CREATE TABLE auditoria (
    id SERIAL PRIMARY KEY,
    tabela_nome VARCHAR(50),
    operacao VARCHAR(20),
    registro_id INTEGER,
    dados_antigos JSONB,
    dados_novos JSONB,
    usuario_modificacao VARCHAR(50),
    data_modificacao TIMESTAMP
);

CREATE OR REPLACE FUNCTION fn_auditoria()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO auditoria (
        tabela_nome,
        operacao,
        registro_id,
        dados_antigos,
        dados_novos,
        usuario_modificacao,
        data_modificacao
    ) VALUES (
        TG_TABLE_NAME,
        TG_OP,
        COALESCE(NEW.id, OLD.id),
        CASE WHEN TG_OP IN ('UPDATE', 'DELETE') THEN to_jsonb(OLD)
        ELSE NULL END,
        CASE WHEN TG_OP IN ('UPDATE', 'INSERT') THEN to_jsonb(NEW)
        ELSE NULL END,
        current_user,
        current_timestamp
    );
    RETURN COALESCE(NEW, OLD);

```

```
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER trg_produtos_auditoria  
AFTER INSERT OR UPDATE OR DELETE ON produtos  
FOR EACH ROW EXECUTE FUNCTION fn_auditoria();
```

Exercício 5: Views Materializadas

Descrição

Crie views materializadas para relatórios de vendas.

Requisitos

- Agregações complexas
- Atualização programada
- Índices para performance

Solução

```
CREATE MATERIALIZED VIEW mv_resumo_vendas AS  
SELECT  
    p.categoria_id,  
    c.nome as categoria,  
    DATE_TRUNC('month', v.data_venda) as mes,  
    COUNT(*) as total_vendas,  
    SUM(v.quantidade) as quantidade_total,  
    SUM(v.valor_total) as valor_total  
FROM historico_vendas v  
JOIN produtos p ON v.producto_id = p.id  
JOIN categorias c ON p.categoria_id = c.id  
GROUP BY p.categoria_id, c.nome, DATE_TRUNC('month', v.data_venda)  
WITH DATA;  
  
CREATE UNIQUE INDEX idx_mv_resumo_vendas
```

```
ON mv_resumo_vendas (categoria_id, mes);

CREATE OR REPLACE FUNCTION refresh_mv_resumo_vendas()
RETURNS void AS $$

BEGIN
    REFRESH MATERIALIZED VIEW CONCURRENTLY mv_resumo_vendas;
END;

$$ LANGUAGE plpgsql;
```

Exercício 6: Schemas e Segurança

Descrição

Configure schemas diferentes para separação de responsabilidades.

Requisitos

- Schemas para diferentes módulos
- Controle de acesso granular
- Roles específicas

Solução

```
-- Criação de schemas
CREATE SCHEMA vendas;
CREATE SCHEMA estoque;
CREATE SCHEMA relatorios;

-- Criação de roles
CREATE ROLE vendedor;
CREATE ROLE estoquista;
CREATE ROLE analista;

-- Permissões
GRANT USAGE ON SCHEMA vendas TO vendedor;
GRANT SELECT, INSERT ON ALL TABLES IN SCHEMA vendas TO vendedor;
```

```
GRANT USAGE ON SCHEMA estoque TO estoquista;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA estoque TO
estoquista;

GRANT USAGE ON SCHEMA relatorios TO analista;
GRANT SELECT ON ALL TABLES IN SCHEMA relatorios TO analista;
```

Exercício 7: Constraints Complexas

Descrição

Implemente constraints avançadas com funções personalizadas.

Requisitos

- Validação complexa de dados
- Constraints exclusivas condicionais
- Constraints entre tabelas

Solução

```
-- Função para validação de estoque
CREATE OR REPLACE FUNCTION fn_validar_estoque()
RETURNS trigger AS $$

BEGIN
    IF NEW.quantidade < 0 THEN
        RAISE EXCEPTION 'Quantidade não pode ser negativa';
    END IF;

    IF NEW.quantidade < NEW.quantidade_minima THEN
        RAISE NOTICE 'Estoque abaixo do mínimo';
    END IF;

    RETURN NEW;
END;
```

```
$$ LANGUAGE plpgsql;

-- Constraint exclusiva condicional
ALTER TABLE produtos
ADD CONSTRAINT uk_produto_codigo
EXCLUDE USING gist (
    nome WITH =,
    codigo_barras WITH =
) WHERE (status = 'ativo');

-- Constraint entre tabelas
ALTER TABLE pedidos
ADD CONSTRAINT check_limite_credito
CHECK (
    valor_total <= (
        SELECT limite_credito
        FROM clientes
        WHERE id = cliente_id
    )
);
```

Exercício 8: Extensões e Tipos Customizados

Descrição

Utilize extensões PostgreSQL e crie tipos personalizados.

Requisitos

- Extensão para geolocalização
- Tipo customizado para endereço
- Tipo customizado para dinheiro

Solução

```

-- Instalação de extensão
CREATE EXTENSION IF NOT EXISTS postgis;

-- Tipo customizado para endereço
CREATE TYPE endereco_type AS (
    logradouro VARCHAR(100),
    numero VARCHAR(10),
    complemento VARCHAR(50),
    bairro VARCHAR(50),
    cidade VARCHAR(50),
    estado CHAR(2),
    cep VARCHAR(8)
);

-- Tipo customizado para moeda
CREATE TYPE moeda_type AS (
    valor DECIMAL(10,2),
    moeda CHAR(3)
);

-- Uso dos tipos
CREATE TABLE lojas (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100),
    endereco endereco_type,
    localizacao geography(Point),
    faturamento_mensal moeda_type
);

```

Exercício 9: Otimização de Schema

Descrição

Otimize a estrutura do banco de dados para performance.

Requisitos

- Índices apropriados
- Estatísticas personalizadas
- Configuração de storage

Solução

```
-- Índices para joins frequentes
CREATE INDEX idx_produtos_categoria_preco
ON produtos (categoria_id, preco);

-- Índice parcial
CREATE INDEX idx_produtos_ativos
ON produtos (nome)
WHERE status = 'ativo';

-- Índice de texto
CREATE INDEX idx_produtos_busca
ON produtos USING gin(to_tsvector('portuguese', nome || ' ' || descricao));

-- Estatísticas personalizadas
CREATE STATISTICS produtos_stats (dependencies)
ON categoria_id, status, preco
FROM produtos;

-- Configuração de storage
ALTER TABLE produtos SET (
    autovacuum_vacuum_scale_factor = 0.1,
    autovacuum_analyze_scale_factor = 0.05
);
```

Exercício 10: Migração de Dados

Descrição

Crie scripts para migração segura de dados.

Requisitos

- Verificação de integridade
- Tratamento de erros
- Rollback em caso de falha

Solução

```
DO $$  
DECLARE  
    v_count INTEGER;  
BEGIN  
    -- Início da transação  
    BEGIN  
        -- Criação de tabela temporária  
        CREATE TEMP TABLE tmp_produtos AS  
        SELECT * FROM produtos_legado;  
  
        GET DIAGNOSTICS v_count = ROW_COUNT;  
        RAISE NOTICE 'Dados copiados: %', v_count;  
  
        -- Validação dos dados  
        PERFORM count(*)  
        FROM tmp_produtos  
        WHERE preco < 0;  
  
        IF FOUND THEN  
            RAISE EXCEPTION 'Dados inválidos encontrados';  
        END IF;  
  
        -- Inserção dos dados  
        INSERT INTO produtos_novo  
        SELECT * FROM tmp_produtos;  
  
        -- Commit se tudo ok  
        RAISE NOTICE 'Migração concluída com sucesso';
```

```
EXCEPTION WHEN OTHERS THEN
    -- Rollback em caso de erro
    RAISE NOTICE 'Erro na migração: %', SQLERRM;
    RAISE;
END;
END;
$$;
```

Critérios de Avaliação

- Implementação correta das funcionalidades
- Uso adequado de recursos avançados
- Performance e escalabilidade
- Tratamento de erros
- Documentação do código

Dicas de Estudo

- Consulte a documentação oficial do PostgreSQL
- Teste diferentes cenários
- Monitore a performance
- Mantenha scripts de rollback
- Documente decisões de design

Exercícios Avançados de DDL

Exercício 1: Sistema de Particionamento

Descrição

Implemente um sistema de particionamento automático para uma tabela de logs.

Requisitos

- Particionamento por data
- Manutenção automática
- Rotação de partições

Solução

```
-- Criar tabela particionada
CREATE TABLE logs (
    id BIGSERIAL,
    timestamp TIMESTAMP NOT NULL,
    nivel VARCHAR(20),
    mensagem TEXT,
    dados JSONB
) PARTITION BY RANGE (timestamp);

-- Função para criar partições automaticamente
CREATE OR REPLACE FUNCTION criar_particao_logs(
    data_inicio DATE,
    data_fim DATE
) RETURNS VOID AS $$
BEGIN
    EXECUTE format(
        'CREATE TABLE IF NOT EXISTS logs_%s
        PARTITION OF logs
        FOR VALUES FROM (%L) TO (%L)',
```

```

        to_char(data_inicio, 'YYYYMM'),
        data_inicio,
        data_fim
    );
END;
$$ LANGUAGE plpgsql;

-- Criar partições para os próximos 12 meses
DO $$

DECLARE
    v_data DATE := date_trunc('month', CURRENT_DATE);
    v_fim DATE;
BEGIN
    FOR i IN 0..11 LOOP
        v_fim := v_data + interval '1 month';
        PERFORM criar_particao_logs(v_data, v_fim);
        v_data := v_fim;
    END LOOP;
END;
$$ LANGUAGE plpgsql;

```

Exercício 2: Gerenciamento de Extensões

Descrição

Desenvolva um sistema de gerenciamento de extensões PostgreSQL.

Requisitos

- Instalação controlada
- Versionamento
- Dependências

Solução

```

-- Schema para gerenciamento de extensões
CREATE SCHEMA ext_manager;

-- Tabela de controle de extensões
CREATE TABLE ext_manager.extensions (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    versao VARCHAR(50) NOT NULL,
    dependencias JSONB,
    instalada_em TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    ultima_atualizacao TIMESTAMP,
    status VARCHAR(20) DEFAULT 'INSTALADA',
    UNIQUE(nome, versao)
);

-- Função para instalar extensão
CREATE OR REPLACE FUNCTION ext_manager.instalar_extensao(
    p_nome VARCHAR,
    p_versao VARCHAR DEFAULT NULL,
    p_deps JSONB DEFAULT NULL
) RETURNS VOID AS $$

BEGIN
    -- Verificar se já está instalada
    IF EXISTS (
        SELECT 1
        FROM ext_manager.extensions
        WHERE nome = p_nome
        AND status = 'INSTALADA'
    ) THEN
        RAISE EXCEPTION 'Extensão % já está instalada', p_nome;
    END IF;

    -- Instalar extensão
    IF p_versao IS NULL THEN
        EXECUTE format('CREATE EXTENSION IF NOT EXISTS %I',
p_nome);
    ELSE

```

```

        EXECUTE format('CREATE EXTENSION IF NOT EXISTS %I VERSION
%L',
                     p_nome, p_versao);
END IF;

-- Registrar instalação
INSERT INTO ext_manager.extensions (
    nome, versao, dependencias
) VALUES (
    p_nome,
    COALESCE(p_versao,
              (SELECT extversion FROM pg_extension WHERE extname =
p_nome)),
    p_deps
);
END;
$$ LANGUAGE plpgsql;

-- Função para atualizar extensão
CREATE OR REPLACE FUNCTION ext_manager.atualizar_extensao(
    p_nome VARCHAR,
    p_versao_nova VARCHAR

```

Exercício 3: Sistema de Versionamento

Descrição

Crie um sistema de versionamento de schema do banco de dados.

Requisitos

- Controle de versões
- Migrations
- Rollback

Solução

```
-- Schema para versionamento
CREATE SCHEMA IF NOT EXISTS versioning;

-- Tabela de controle de versões
CREATE TABLE versioning.schema_versions (
    version_id SERIAL PRIMARY KEY,
    version_number VARCHAR(50) NOT NULL,
    applied_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    description TEXT,
    script_name VARCHAR(200),
    checksum VARCHAR(64),
    applied_by VARCHAR(100) DEFAULT CURRENT_USER,
    status VARCHAR(20) DEFAULT 'SUCCESS',
    UNIQUE(version_number)
);

-- Função para aplicar versão
CREATE OR REPLACE FUNCTION versioning.apply_version(
    p_version VARCHAR,
    p_description TEXT,
    p_script TEXT
) RETURNS VOID AS $$

DECLARE
    v_checksum VARCHAR(64);
BEGIN
    -- Calcular checksum
    v_checksum := encode(sha256(p_script::bytea), 'hex');

    -- Registrar versão
    INSERT INTO versioning.schema_versions (
        version_number,
        description,
        script_name,
        checksum
    ) VALUES (

```

```

    p_version,
    p_description,
    p_script,
    v_checksum
);

-- Executar script
EXECUTE p_script;

EXCEPTION WHEN OTHERS THEN
    -- Marcar como falha em caso de erro
    UPDATE versioning.schema_versions
    SET status = 'FAILED',
        description = description || E'\nError: ' || SQLERRM
    WHERE version_number = p_version;

    RAISE;
END;
$$ LANGUAGE plpgsql;

```

Exercício 4: Auditoria Avançada

Descrição

Implemente um sistema de auditoria completo com histórico de alterações.

Requisitos

- Rastreamento de todas as alterações DDL
- Histórico de modificações de dados
- Sistema de recuperação

Solução

```

-- Schema para auditoria
CREATE SCHEMA audit;

```

```

-- Tabela de eventos DDL
CREATE TABLE audit.ddl_events (
    id BIGSERIAL PRIMARY KEY,
    evento_timestamp TIMESTAMP WITH TIME ZONE DEFAULT
CURRENT_TIMESTAMP,
    usuario_bd NAME DEFAULT CURRENT_USER,
    usuario_app VARCHAR(50),
    tipo_comando VARCHAR(20),
    objeto_schema VARCHAR(100),
    objeto_nome VARCHAR(100),
    comando_sql TEXT
);

-- Função para capturar eventos DDL
CREATE OR REPLACE FUNCTION audit.log_ddl_event()
RETURNS event_trigger AS $$

DECLARE
    obj record;
    comando TEXT;

BEGIN
    SELECT * INTO obj FROM pg_event_trigger_ddl_commands() LIMIT
1;

    SELECT current_query() INTO comando;

    INSERT INTO audit.ddl_events (
        tipo_comando,
        objeto_schema,
        objeto_nome,
        comando_sql
    ) VALUES (
        obj.command_tag,
        obj.schema_name,
        obj.object_name,
        comando
    );
END;

```

```
$$ LANGUAGE plpgsql;

-- Event trigger para DDL
CREATE EVENT TRIGGER trg_audit_ddl
ON ddl_command_end
EXECUTE FUNCTION audit.log_ddl_event();
```

Critérios de Avaliação

1. Robustez

- Tratamento de erros
- Validações
- Consistência dos dados

2. Performance

- Eficiência das operações
- Uso de recursos
- Escalabilidade

3. Manutenibilidade

- Clareza do código
- Documentação
- Modularidade

4. Segurança

- Controle de acesso
- Proteção dos dados
- Auditoria

5. Funcionalidade

- Atendimento aos requisitos
- Completude da solução
- Integração com sistemas existentes

Recursos Adicionais

Documentação

- PostgreSQL Partitioning (<https://www.postgresql.org/docs/current/ddl-partitioning.html>)
- PostgreSQL Replication (<https://www.postgresql.org/docs/current/runtime-config-replication.html>)
- PostgreSQL Extensions (<https://www.postgresql.org/docs/current/extend-extensions.html>)

Ferramentas Recomendadas

- pgAdmin 4
- DBeaver
- pg_partman
- pg_audit

Boas Práticas

CHECKLIST DE IMPLEMENTAÇÃO:

- ✓ Testes de performance
- ✓ Scripts de rollback
- ✓ Documentação técnica
- ✓ Monitoramento

Próximos Passos

1. Prática Avançada

- Implementar soluções em ambiente de teste
- Realizar testes de carga
- Simular cenários de falha

2. Aprofundamento

- Estudar casos de uso específicos
- Explorar recursos avançados
- Participar de comunidades

3. Projetos Práticos

- Desenvolver soluções completas
- Documentar implementações
- Compartilhar experiências

Conclusão

Os exercícios avançados de DDL apresentados neste módulo cobrem aspectos críticos do desenvolvimento de bancos de dados em nível empresarial. A prática consistente destes conceitos é fundamental para o domínio das técnicas avançadas de gerenciamento de dados.

Referências

1. PostgreSQL Documentation

2. Database Design Best Practices
3. Enterprise Database Architecture
4. Performance Tuning Guidelines



Nota: Mantenha este documento como referência para implementações futuras e continue praticando os conceitos apresentados.

Laboratórios de DDL

Visão Geral

Os laboratórios práticos de DDL (Data Definition Language) são projetados para fornecer experiência hands-on com operações de definição de dados em ambientes PostgreSQL. Cada laboratório apresenta cenários reais e desafios práticos comumente encontrados em ambientes de produção.

Estrutura dos Laboratórios

Cada laboratório segue uma estrutura consistente:

1. Preparação do Ambiente

- Configuração inicial
- Pré-requisitos
- Scripts de setup

2. Objetivos de Aprendizado

- Conceitos principais
- Habilidades técnicas
- Resultados esperados

3. Roteiro Prático

- Instruções passo a passo
- Comandos e scripts
- Pontos de verificação

4. Avaliação

- Critérios de conclusão

- Testes de validação
- Métricas de sucesso

Ambiente de Laboratório

```
-- Criar database dedicado para laboratórios
CREATE DATABASE lab_ddl;

-- Schema para isolamento de exercícios
CREATE SCHEMA lab_workspace;

-- Tabela de controle de progresso
CREATE TABLE lab_workspace.lab_progress (
    lab_id SERIAL PRIMARY KEY,
    lab_name VARCHAR(100),
    start_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    completion_time TIMESTAMP,
    status VARCHAR(20) DEFAULT 'IN_PROGRESS',
    notes TEXT
);
```

Laboratórios Disponíveis

1. Criação de Database

- Estruturação inicial de databases
- Configurações básicas
- Boas práticas de nomenclatura

2. Evolução de Schema

- Alterações estruturais
- Migrações seguras

- Versionamento de schema

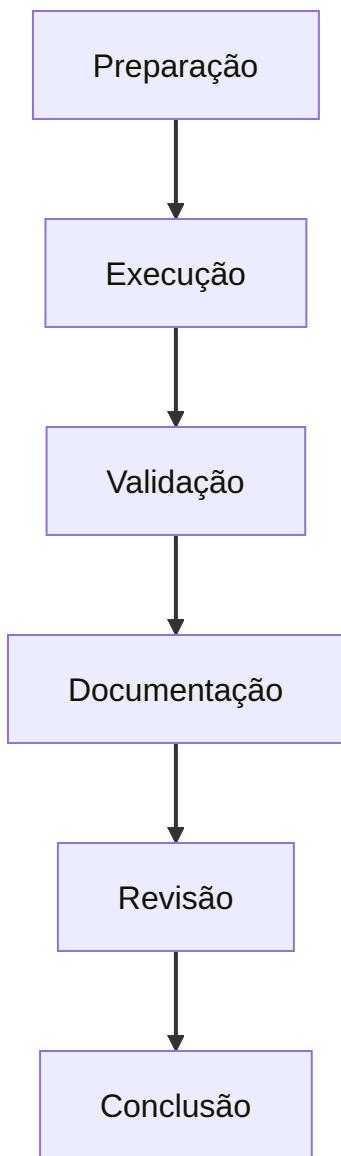
3. Gerenciamento de Constraints

- Implementação de regras de negócio
- Integridade referencial
- Validações customizadas

4. Manutenção de Tabelas

- Operações de manutenção
- Otimização de estruturas
- Gerenciamento de espaço

Fluxo de Trabalho Recomendado



Boas Práticas

DIRETRIZES DE LABORATÓRIO:	
✓ Sempre fazer backup antes das práticas	
✓ Documentar todas as alterações	
✓ Testar em ambiente isolado	
✓ Validar resultados	
✓ Manter scripts de rollback	

Ferramentas Necessárias

1. PostgreSQL Client

- psql
- pgAdmin 4
- DBeaver

2. Utilitários

- pg_dump
- pg_restore
- pg_bench

3. Scripts de Suporte

```
-- Script de verificação de ambiente
CREATE OR REPLACE FUNCTION lab_workspace.check_environment()
RETURNS TABLE (
    check_item VARCHAR,
    status VARCHAR,
    details TEXT
) AS $$

BEGIN
    RETURN QUERY
        SELECT 'Database Version'::VARCHAR,
               version()::VARCHAR,
               'Verificação da versão do PostgreSQL '::TEXT
        UNION ALL
        SELECT 'Available Space',
               pg_size_pretty(pg_database_size(current_database())),
               'Espaço utilizado pelo database atual';
END;
$$ LANGUAGE plpgsql;
```

Navegação dos Laboratórios

- Laboratório de Criação de Database ([Laboratório: Criação de Database](#))
- Laboratório de Evolução de Schema ([Laboratório: Evolução de Schema](#))
- Laboratório de Gerenciamento de Constraints ([Laboratório: Gerenciamento de Constraints](#))
- Laboratório de Manutenção de Tabelas ([Laboratório: Manutenção de Tabelas](#))

Supporte e Recursos

Documentação

- PostgreSQL Official Documentation (<https://www.postgresql.org/docs/>)
- DDL Reference Guide (<https://www.postgresql.org/docs/current/ddl.html>)
- Schema Management Best Practices (<https://www.postgresql.org/docs/current/ddl-schemas.html>)

Comunidade

- Fórum PostgreSQL
- Stack Overflow
- GitHub Discussions

Conclusão

Os laboratórios DDL fornecem uma base prática essencial para o desenvolvimento de habilidades em gerenciamento de estruturas de dados. A prática regular destes exercícios contribuirá significativamente para sua expertise em administração de bancos de dados PostgreSQL.

Próximos Passos

1. Preparação

- Configure seu ambiente local
- Revise os pré-requisitos
- Familiarize-se com as ferramentas

2. Execução

- Siga os laboratórios em ordem
- Complete todos os exercícios
- Documente seus resultados

3. Avançado

- Explore variações dos exercícios
- Crie seus próprios cenários
- Compartilhe experiências



Nota: Certifique-se de manter backups e usar ambientes de teste apropriados durante a execução dos laboratórios.

Laboratório: Criação de Database

Objetivo

Praticar a criação e configuração de databases PostgreSQL, aplicando boas práticas e explorando diferentes opções de configuração.

Pré-requisitos

- PostgreSQL 12 ou superior instalado
- Acesso administrativo ao servidor
- psql ou ferramenta similar de acesso

Exercícios

1. Database Básico

```
-- Criar database simples
CREATE DATABASE lab_basic;

-- Verificar criação
SELECT datname, encoding, datcollate, datctype
FROM pg_database
WHERE datname = 'lab_basic';
```

2. Database com Configurações Específicas

```
-- Criar database com configurações customizadas
CREATE DATABASE lab_advanced
WITH
ENCODING = 'UTF8'
LC_COLLATE = 'pt_BR.UTF-8'
LC_CTYPE = 'pt_BR.UTF-8'
TEMPLATE = template0
```

```
OWNER = lab_user  
CONNECTION LIMIT = 100;
```

3. Database com Tablespaces

```
-- Criar tablespace customizado  
CREATE TABLESPACE lab_space  
    LOCATION '/path/to/data/lab_tablespace';  
  
-- Criar database usando tablespace  
CREATE DATABASE lab_distributed  
    TABLESPACE lab_space;
```

Tarefas Práticas

Tarefa 1: Setup Inicial

1. Criar um novo database para ambiente de desenvolvimento
2. Configurar encoding e locale apropriados
3. Estabelecer limites de conexão
4. Verificar configurações

Tarefa 2: Configuração Avançada

1. Implementar tablespaces separados para dados e índices
2. Configurar parâmetros de performance
3. Estabelecer políticas de backup

Tarefa 3: Migração

1. Criar database template
2. Clonar estrutura para novo database

3. Verificar integridade

Verificações

```
-- Script de verificação de configurações
CREATE OR REPLACE FUNCTION check_database_config()
RETURNS TABLE (
    parameter_name VARCHAR,
    current_value VARCHAR,
    recommended_value VARCHAR
) AS $$

BEGIN
    RETURN QUERY
    SELECT
        'max_connections'::VARCHAR,
        current_setting('max_connections')::VARCHAR,
        '100'::VARCHAR
    UNION ALL
    SELECT
        'default_transaction_isolation',
        current_setting('default_transaction_isolation'),
        'read committed';
END;
$$ LANGUAGE plpgsql;
```

Boas Práticas

CHECKLIST DE CRIAÇÃO:

- ✓ Encoding adequado
- ✓ Locale correto
- ✓ Permissões apropriadas
- ✓ Limites de conexão
- ✓ Tablespaces configurados

Troubleshooting

Problemas Comuns

1. Erro de Locale

```
-- Solução: Usar template0
CREATE DATABASE lab_fix
    TEMPLATE template0
    LC_COLLATE 'pt_BR.UTF-8'
    LC_CTYPE 'pt_BR.UTF-8';
```

2. Limite de Conexões

```
-- Ajustar limite
ALTER DATABASE lab_fix
    CONNECTION LIMIT 150;
```

3. Permissões

```
-- Corrigir ownership
ALTER DATABASE lab_fix
    OWNER TO lab_admin;
```

Scripts de Suporte

Script de Validação

```
CREATE OR REPLACE PROCEDURE validate_database_creation(
    db_name VARCHAR
) AS $$
DECLARE
    v_exists BOOLEAN;
BEGIN
    SELECT EXISTS (
        SELECT 1
```

```

        FROM pg_database
        WHERE datname = db_name
    ) INTO v_exists;

    IF NOT v_exists THEN
        RAISE EXCEPTION 'Database % não existe', db_name;
    END IF;

    -- Verificar configurações básicas
    PERFORM check_database_config();
END;
$$ LANGUAGE plpgsql;

```

Exercícios Avançados

1. Replicação

- Configurar database para replicação
- Implementar streaming replication
- Testar failover

2. Performance

- Otimizar configurações de memória
- Ajustar parâmetros de vacuum
- Monitorar desempenho

3. Segurança

- Implementar criptografia
- Configurar SSL
- Estabelecer políticas de acesso

Critérios de Conclusão

1. Funcionalidade

- Database criado corretamente
- Configurações aplicadas
- Conexões funcionando

2. Performance

- Tempos de resposta aceitáveis
- Uso de recursos otimizado
- Monitoramento estabelecido

3. Segurança

- Permissões corretas
- Acessos controlados
- Logs configurados

Próximos Passos

1. Explorar configurações avançadas
2. Implementar monitoramento
3. Documentar procedimentos
4. Automatizar processos

Recursos Adicionais

- PostgreSQL Database Creation (<https://www.postgresql.org/docs/current/sql-createdatabase.html>)

- Database Configuration (<https://www.postgresql.org/docs/current/runtime-config.html>)
- Tablespace Management (<https://www.postgresql.org/docs/current/manage-ag-tablespaces.html>)



Nota: Mantenha este documento como referência para futuras criações de databases e configurações avançadas.

Laboratório: Evolução de Schema

Objetivo

Praticar alterações de schema em um banco de dados em produção, aplicando boas práticas de versionamento e migração.

Cenário

Você é o DBA responsável por um sistema de e-commerce em crescimento que precisa evoluir seu schema para acomodar novos requisitos de negócio.

Setup Inicial

```
-- Schema inicial
CREATE TABLE produtos (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    preco DECIMAL(10,2) NOT NULL,
    estoque INT DEFAULT 0
);

CREATE TABLE clientes (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL
);

CREATE TABLE pedidos (
    id SERIAL PRIMARY KEY,
    cliente_id INT REFERENCES clientes(id),
    data_pedido TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Exercícios

1. Adição de Novas Features

```
-- TODO: Adicionar suporte para categorias de produtos
-- 1.1 Crie uma tabela de categorias
-- 1.2 Adicione uma coluna categoria_id em produtos
-- 1.3 Estabeleça a foreign key apropriada
```

2. Modificação de Estrutura Existente

```
-- TODO: Expandir informações de cliente
-- 2.1 Adicione colunas para telefone e endereço
-- 2.2 Torne o telefone único e não nulo
-- 2.3 Migre dados existentes adequadamente
```

3. Gestão de Constraints

```
-- TODO: Implementar regras de negócio
-- 3.1 Adicione check constraint para preço mínimo
-- 3.2 Crie unique constraint para SKU de produtos
-- 3.3 Implemente trigger para log de alterações
```

Desafios Extras

1. Versionamento

```
-- Implemente um sistema de controle de versão de schema
CREATE TABLE schema_version (
    version INT PRIMARY KEY,
    applied_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    description TEXT
);

-- TODO: Crie scripts de up/down para cada alteração
```

2. Zero Downtime

Desenvolva estratégias para:

- Adição de colunas não-nulas
- Renomeação de colunas
- Divisão de tabelas

3. Rollback Plan

```
-- TODO: Para cada alteração, crie um plano de rollback
-- Exemplo:
BEGIN;
    -- Alterações
    ALTER TABLE produtos ADD COLUMN categoria_id INT;

    -- Verificação
    SELECT COUNT(*) FROM produtos WHERE categoria_id IS NULL;

    -- Decisão de commit/rollback
    -- COMMIT ou ROLLBACK;
END;
```

Critérios de Avaliação

1. Segurança de Dados

- Backup antes das alterações
- Validação de integridade
- Tratamento de erros

2. Performance

- Tempo de execução
- Impacto em produção
- Uso de recursos

3. Manutenibilidade

- Documentação
- Versionamento
- Scripts de rollback

Entrega

Crie um arquivo SQL contendo:

1. Scripts de alteração
2. Scripts de rollback
3. Documentação das mudanças
4. Logs de teste

Dicas

CHECKLIST DE ALTERAÇÕES:

- Backup realizado?
- Scripts testados em homologação?
- Plano de rollback preparado?
- Janela de manutenção definida?
- Stakeholders notificados?

Recursos Adicionais

- Documentação PostgreSQL sobre ALTER
(<https://www.postgresql.org/docs/current/sql-alter.html>)
- Práticas de Schema Migration
(<https://flywaydb.org/documentation/concepts/migrations>)

- Zero Downtime Deployments (<https://www.postgresql.org/docs/current/monitoring-stats.html>)

Próximos Passos

1. Implemente monitoramento de performance
2. Automatize testes de migração
3. Crie documentação detalhada
4. Prepare apresentação dos resultados

Laboratório: Gerenciamento de Constraints

Objetivo

Praticar a implementação e gerenciamento de diferentes tipos de constraints no PostgreSQL, garantindo a integridade dos dados e regras de negócio.

Ambiente de Teste

```
-- Schema para o laboratório
CREATE SCHEMA lab_constraints;

-- Tabelas para prática
CREATE TABLE lab_constraints.produtos (
    id SERIAL PRIMARY KEY,
    codigo VARCHAR(20) UNIQUE,
    nome VARCHAR(100) NOT NULL,
    preco DECIMAL(10, 2),
    categoria_id INTEGER,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE lab_constraints.categorias (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(50) UNIQUE,
    ativa BOOLEAN DEFAULT true
);
```

Exercícios Práticos

1. Constraints Básicas

```
-- Primary Key
ALTER TABLE lab_constraints.produtos
```

```

    ADD CONSTRAINT pk_produtos PRIMARY KEY (id);

-- Unique Constraint
ALTER TABLE lab_constraints.produtos
    ADD CONSTRAINT uk_produtos_codigo UNIQUE (codigo);

-- Not Null
ALTER TABLE lab_constraints.produtos
    ALTER COLUMN nome SET NOT NULL;

```

2. Foreign Keys

```

-- Adicionar Foreign Key com diferentes ações
ALTER TABLE lab_constraints.produtos
    ADD CONSTRAINT fk_produtos_categoria
        FOREIGN KEY (categoria_id)
        REFERENCES lab_constraints.categorias(id)
        ON DELETE RESTRICT
        ON UPDATE CASCADE;

```

3. Check Constraints

```

-- Validação de preço
ALTER TABLE lab_constraints.produtos
    ADD CONSTRAINT ck_produtos_preco
        CHECK (preco > 0);

-- Validação de código
ALTER TABLE lab_constraints.produtos
    ADD CONSTRAINT ck_produtos_codigo
        CHECK (codigo ~ '^[A-Z]{2}\d{4}$');

```

4. Exclusion Constraints

```

-- Criar extensão btree_gist se necessário
CREATE EXTENSION IF NOT EXISTS btree_gist;

```

```
-- Exemplo de constraint de exclusão
CREATE TABLE lab_constraints.eventos (
    id SERIAL PRIMARY KEY,
    sala_id INTEGER,
    periodo tsrange,
    EXCLUDE USING gist (
        sala_id WITH =,
        periodo WITH &&
    )
);
```

Tarefas Práticas

Tarefa 1: Implementação de Regras de Negócio

```
-- Regra: Produto não pode ser deletado se tiver vendas
CREATE TABLE lab_constraints.vendas (
    id SERIAL PRIMARY KEY,
    produto_id INTEGER,
    quantidade INTEGER,
    CONSTRAINT fk_vendas_produto
    FOREIGN KEY (produto_id)
    REFERENCES lab_constraints.produtos(id)
    ON DELETE RESTRICT
);

-- Regra: Quantidade deve ser positiva
ALTER TABLE lab_constraints.vendas
    ADD CONSTRAINT ck_vendas_quantidade
    CHECK (quantidade > 0);
```

Tarefa 2: Constraints Dinâmicas

```
-- Função para validação customizada
CREATE OR REPLACE FUNCTION lab_constraints.validar_preco()
```

```

RETURNS trigger AS $$

BEGIN
    IF NEW.preco < (
        SELECT MIN(preco)
        FROM lab_constraints.produtos
        WHERE categoria_id = NEW.categoria_id
    ) * 0.5 THEN
        RAISE EXCEPTION 'Preço muito baixo para a categoria';
    END IF;
    RETURN NEW;
END;

$$ LANGUAGE plpgsql;

-- Trigger para aplicar validação
CREATE TRIGGER trg_validar_preco
    BEFORE INSERT OR UPDATE ON lab_constraints.produtos
    FOR EACH ROW
    EXECUTE FUNCTION lab_constraints.validar_preco();

```

Tarefa 3: Gerenciamento de Constraints

```

-- Desabilitar temporariamente
ALTER TABLE lab_constraints.produtos
    DISABLE TRIGGER trg_validar_preco;

-- Reabilitar
ALTER TABLE lab_constraints.produtos
    ENABLE TRIGGER trg_validar_preco;

-- Remover constraint
ALTER TABLE lab_constraints.produtos
    DROP CONSTRAINT IF EXISTS ck_produtos_preco;

```

Verificação de Constraints

```

-- Função para listar todas as constraints
CREATE OR REPLACE FUNCTION lab_constraints.listar_constraints()
RETURNS TABLE (
    tabela VARCHAR,
    constraint_name VARCHAR,
    constraint_type VARCHAR,
    definition TEXT
) AS $$

BEGIN
    RETURN QUERY
    SELECT
        tc.table_schema || '.' || tc.table_name::VARCHAR,
        tc.constraint_name::VARCHAR,
        tc.constraint_type::VARCHAR,
        pg_get_constraintdef(pgc.oid)::TEXT
    FROM information_schema.table_constraints tc
    JOIN pg_constraint pgc
        ON tc.constraint_name = pgc.conname
    WHERE tc.table_schema = 'lab_constraints'
    ORDER BY tc.table_name, tc.constraint_name;
END;

$$ LANGUAGE plpgsql;

```

Boas Práticas

- | |
|--|
| <ul style="list-style-type: none"> DIRETRIZES DE CONSTRAINTS: ✓ Nomes descritivos ✓ Documentação clara ✓ Performance considerada ✓ Validações apropriadas ✓ Manutenção planejada |
|--|

Troubleshooting

Problemas Comuns

1. Violação de Constraint

```
-- Verificar dados violando a constraint
SELECT * FROM lab_constraints.produtos
WHERE preco <= 0;
```

2. Deadlocks em Foreign Keys

```
-- Ajustar isolamento
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

3. Performance

```
-- Analisar impacto
EXPLAIN ANALYZE
SELECT * FROM lab_constraints.produtos
WHERE codigo = 'AB1234';
```

Scripts de Manutenção

```
-- Verificar integridade
CREATE OR REPLACE PROCEDURE
lab_constraints.verificar_integridade()
AS $$$
DECLARE
    v_constraint RECORD;
BEGIN
    FOR v_constraint IN (
        SELECT * FROM lab_constraints.listar_constraints()
    ) LOOP
        EXECUTE 'VALIDATE CONSTRAINT ' ||
v_constraint.constraint_name ||
' ON ' || v_constraint.tabela;
    END LOOP;

```

```
END;  
$$ LANGUAGE plpgsql;
```

Critérios de Avaliação

1. Implementação

- Constraints corretamente definidas
- Regras de negócio implementadas
- Validações funcionando

2. Performance

- Impacto aceitável
- Índices apropriados
- Otimizações aplicadas

3. Manutenibilidade

- Documentação clara
- Nomes consistentes
- Scripts de manutenção

Próximos Passos

1. Explorar constraints avançadas
2. Implementar casos complexos
3. Otimizar performance
4. Automatizar testes

Recursos Adicionais

- PostgreSQL Constraints (<https://www.postgresql.org/docs/current/ddl-constraints.html>)
- Check Constraints (<https://www.postgresql.org/docs/current/ddl-constraints.html#DDL-CONSTRAINTS-CHECK-CONSTRAINTS>)
- Exclusion Constraints (<https://www.postgresql.org/docs/current/ddl-constraints.html#DDL-CONSTRAINTS-EXCLUSION>)



Nota: Este laboratório deve ser executado em um ambiente de teste para evitar impactos em dados de produção.

Laboratório: Manutenção de Tabelas

Objetivo

Praticar operações de manutenção em tabelas de banco de dados, incluindo otimização de espaço, reorganização de dados e monitoramento de performance.

Cenário

Você é o DBA responsável por um sistema que está em produção há vários anos. As tabelas cresceram significativamente e começaram a apresentar problemas de performance e fragmentação.

Setup Inicial

```
-- Criar tabela de exemplo com fragmentação
CREATE TABLE lab_maintenance_pedidos (
    id SERIAL PRIMARY KEY,
    cliente_id INTEGER,
    produto_id INTEGER,
    quantidade INTEGER,
    valor DECIMAL(10,2),
    status VARCHAR(20),
    data_pedido TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    data_atualizacao TIMESTAMP
);

-- Inserir dados de exemplo (muitos registros)
INSERT INTO lab_maintenance_pedidos (cliente_id, produto_id,
quantidade, valor, status)
SELECT
    floor(random() * 1000)::int,
    floor(random() * 500)::int,
    floor(random() * 10)::int + 1,
```

```

(random() * 1000)::decimal(10,2),
(ARRAY[ 'PENDENTE', 'APROVADO', 'ENVIADO', 'ENTREGUE',
'CANCELADO'])[floor(random() * 5 + 1)]
FROM generate_series(1, 500000);

-- Simular atualizações frequentes para criar fragmentação
UPDATE lab_maintenance.pedidos
SET status = 'APROVADO', data_atualizacao = CURRENT_TIMESTAMP
WHERE id % 3 = 0;

UPDATE lab_maintenance.pedidos
SET status = 'ENVIADO', data_atualizacao = CURRENT_TIMESTAMP
WHERE id % 5 = 0;

```

Exercícios

1. Análise de Fragmentação

```

-- Verificar estatísticas da tabela
SELECT
    schemaname,
    relname,
    n_live_tup,
    n_dead_tup,
    last_vacuum,
    last_autovacuum
FROM pg_stat_user_tables
WHERE relname = 'pedidos';

-- Verificar bloat (inchamento) da tabela
-- Requer extensão pgstattuple
CREATE EXTENSION IF NOT EXISTS pgstattuple;

SELECT * FROM pgstattuple('lab_maintenance.pedidos');

```

2. Operações de VACUUM

```

-- VACUUM básico
VACUUM lab_maintenance.pedidos;

-- VACUUM FULL (requer lock exclusivo)
VACUUM FULL lab_maintenance.pedidos;

-- VACUUM ANALYZE
VACUUM ANALYZE lab_maintenance.pedidos;

-- Verificar efeito do VACUUM
SELECT
    schemaname,
    relname,
    n_live_tup,
    n_dead_tup,
    last_vacuum
FROM pg_stat_user_tables
WHERE relname = 'pedidos';

```

3. Reconstrução de Índices

```

-- Criar índices para a tabela
CREATE INDEX idx_pedidos_cliente ON
lab_maintenance.pedidos(cliente_id);
CREATE INDEX idx_pedidos_status ON
lab_maintenance.pedidos(status);
CREATE INDEX idx_pedidos_data ON
lab_maintenance.pedidos(data_pedido);

-- Verificar fragmentação de índices
SELECT
    indexrelname,
    idx_scan,
    idx_tup_read,
    idx_tup_fetch
FROM pg_stat_user_indexes
WHERE relname = 'pedidos';

```

```
-- Reconstruir índice
REINDEX INDEX idx_pedidos_cliente;

-- Reconstruir todos os índices da tabela
REINDEX TABLE lab_maintenance.pedidos;
```

4. Monitoramento de Espaço

```
-- Verificar tamanho da tabela e índices
SELECT

pg_size.pretty(pg_total_relation_size('lab_maintenance.pedidos')) as total_size,
    pg_size.pretty(pg_relation_size('lab_maintenance.pedidos')) as table_size,
    pg_size.pretty(pg_indexes_size('lab_maintenance.pedidos')) as indexes_size;

-- Verificar espaço por schema
SELECT
    schemaname,
    pg_size.pretty(sum(pg_relation_size(schemaname || '.' || relname))) as size
FROM pg_stat_user_tables
GROUP BY schemaname
ORDER BY sum(pg_relation_size(schemaname || '.' || relname)) DESC;
```

Tarefas Práticas

Tarefa 1: Otimização de Tabela Grande

1. Crie uma tabela com pelo menos 1 milhão de registros
2. Insira, atualize e exclua registros para criar fragmentação
3. Analise o estado da tabela

4. Aplique as operações de manutenção apropriadas

5. Compare o desempenho antes e depois

Tarefa 2: Automação de Manutenção

```
-- Criar função para manutenção automática
CREATE OR REPLACE FUNCTION lab_maintenance.manter_tabela(tabela
text)
RETURNS void AS $$

BEGIN
    EXECUTE 'VACUUM ANALYZE ' || tabela;
    RAISE NOTICE 'Manutenção concluída para %', tabela;
END;

$$ LANGUAGE plpgsql;

-- Agendar manutenção (simulação)
SELECT lab_maintenance.manter_tabela('lab_maintenance.pedidos');
```

Boas Práticas

CHECKLIST DE MANUTENÇÃO:

- Monitorar crescimento de tabelas
- Configurar autovacuum apropriadamente
- Agendar VACUUM ANALYZE regularmente
- Reconstruir índices periodicamente
- Monitorar bloat de tabelas e índices
- Manter estatísticas atualizadas

Considerações de Performance

1. Janelas de Manutenção

- Agendar operações pesadas para períodos de baixo uso

- Considerar impacto em sistemas 24/7
- Balancear frequência e duração

2. Configurações do Autovacuum

```
-- Verificar configurações atuais  
SHOW autovacuum_vacuum_threshold;  
SHOW autovacuum_analyze_threshold;  
SHOW autovacuum_vacuum_scale_factor;  
SHOW autovacuum_analyze_scale_factor;
```

3. Monitoramento Contínuo

- Implementar alertas para tabelas problemáticas
- Acompanhar tendências de crescimento
- Identificar padrões de acesso

Recursos Adicionais

- PostgreSQL VACUUM Documentation (<https://www.postgresql.org/docs/current/sql-vacuum.html>)
- PostgreSQL Index Maintenance (<https://www.postgresql.org/docs/current/sql-reindex.html>)
- Monitoring Database Activity (<https://www.postgresql.org/docs/current/monitoring-stats.html>)



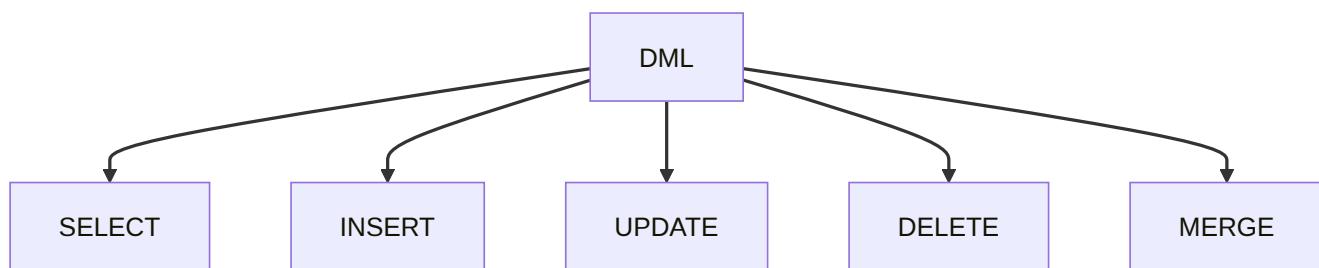
Nota: Este laboratório deve ser executado em um ambiente de teste para evitar impactos em dados de produção.

DML - Linguagem de Manipulação de Dados

|| DATA_MANIPULATOR >> Operações com Dados
|| INSTRUTOR: DML_MASTER

Visão Geral

DML_MASTER apresenta: "A Linguagem de Manipulação de Dados (DML) é o conjunto de comandos SQL usado para gerenciar dados em um banco de dados."



Comandos Principais

1. SELECT

DATA_READER explica: "Recupera dados do banco"

```
-- Consulta básica
SELECT nome, preco
FROM produtos
WHERE categoria = 'Eletrônicos';

-- Com joins
SELECT c.nome, p.descricao
FROM clientes c
JOIN pedidos p ON c.id = p.cliente_id;
```

2. INSERT

DATA_WRITER demonstra: "Adiciona novos registros"

```
-- Inserção simples
INSERT INTO produtos (nome, preco)
VALUES ('Smartphone', 999.99);

-- Inserção múltipla
INSERT INTO produtos (nome, preco)
VALUES
('Tablet', 499.99),
('Notebook', 1499.99);
```

3. UPDATE

DATA_MODIFIER apresenta: "Modifica registros existentes"

```
-- Atualização simples
UPDATE produtos
SET preco = preco * 1.1
WHERE categoria = 'Eletrônicos';

-- Atualização com join
UPDATE pedidos p
SET status = 'Aprovado'
FROM clientes c
WHERE p.cliente_id = c.id
AND c.vip = true;
```

4. DELETE

DATA_REMOVER alerta: "Remove registros do banco"

```
-- Remoção simples
DELETE FROM logs
WHERE data < CURRENT_DATE - INTERVAL '30 days';

-- Remoção com subquery
```

```
DELETE FROM produtos
WHERE id IN (
    SELECT produto_id
    FROM estoque
    WHERE quantidade = 0
);
```

5. MERGE

DATA_SYNC_MASTER explica: "Sincroniza dados entre tabelas"

```
-- Merge básico
MERGE INTO produtos_destino d
USING produtos_origem o
ON (d.id = o.id)
WHEN MATCHED THEN
    UPDATE SET preco = o.preco
WHEN NOT MATCHED THEN
    INSERT (id, nome, preco)
    VALUES (o.id, o.nome, o.preco);
```

Boas Práticas

BEST_PRACTICE_GURU compartilha:

1. Performance

- Use índices apropriados
- Evite SELECT *
- Otimize JOINs
- Limite resultados quando possível

2. Segurança

- Use transações

- Valide dados de entrada
- Faça backup antes de operações grandes
- Use WHERE em UPDATE/DELETE

3. Manutenção

- Documente queries complexas
- Use aliases descritivos
- Mantenha consistência no estilo
- Implemente logging quando necessário

|| CHECKLIST DML:

- || Índices verificados?
- || Transação necessária?
- || WHERE clause adequada?
- || Performance otimizada?
- || Backup realizado?

Padrões Comuns

PATTERN_MASTER apresenta padrões úteis:

1. Upsert

```
INSERT INTO produtos (id, nome, preco)
VALUES (1, 'Smartphone', 999.99)
ON CONFLICT (id)
DO UPDATE SET preco = EXCLUDED.preco;
```

2. Soft Delete

```
-- Em vez de DELETE
UPDATE usuarios
SET ativo = false,
    data_inativacao = CURRENT_TIMESTAMP
WHERE id = 123;
```

3. Batch Processing

```
-- Processamento em lotes
WITH batch AS (
    SELECT id
    FROM pedidos
    WHERE status = 'Pendente'
    LIMIT 1000
    FOR UPDATE SKIP LOCKED
)
UPDATE pedidos p
SET status = 'Processando'
FROM batch b
WHERE p.id = b.id;
```

Troubleshooting

ERROR_HANDLER apresenta soluções para problemas comuns:

1. Deadlocks

```
-- Verificando bloqueios
SELECT blocked_locks.pid AS blocked_pid,
       blocking_locks.pid AS blocking_pid
FROM pg_catalog.pg_locks blocked_locks
JOIN pg_catalog.pg_locks blocking_locks
  ON blocking_locks.locktype = blocked_locks.locktype;
```

2. Performance Issues

```
-- Analisando query
EXPLAIN ANALYZE
SELECT * FROM produtos p
JOIN categorias c ON p.categoria_id = c.id
WHERE p.preco > 100;
```

Conclusão

DML_MASTER conclui: "O domínio dos comandos DML é essencial para qualquer desenvolvedor de banco de dados. Use-os com sabedoria e sempre considere performance e segurança."



Dica Final: Mantenha um ambiente de testes para praticar operações DML complexas antes de executá-las em produção.

SELECT: Recuperando Dados

```
|| QUERY_MASTER >> Arte da Consulta de Dados
|| INSTRUTOR: SELECT_SPECIALIST
```

Fundamentos do SELECT

SELECT_SPECIALIST explica: "O comando SELECT é a base para recuperação de dados em SQL."

Sintaxe Básica

```
SELECT coluna1, coluna2
FROM tabela
WHERE condição
GROUP BY coluna
HAVING condição_grupo
ORDER BY coluna;
```

Cláusulas Principais

1. FROM

DATA_SOURCE_EXPERT demonstra: "Define a fonte dos dados"

```
-- Tabela única
FROM produtos

-- Múltiplas tabelas
FROM produtos p, categorias c

-- Subquery
FROM (SELECT * FROM vendas WHERE ano = 2023) v
```

2. WHERE

FILTER_MASTER apresenta: "Filtrar os registros"

```
-- Comparações básicas
WHERE preco > 100
AND categoria = 'Eletrônicos'

-- Operadores IN/BETWEEN
WHERE status IN ('Ativo', 'Pendente')
AND data_criacao BETWEEN '2023-01-01' AND '2023-12-31'

-- Pattern matching
WHERE nome LIKE 'A%'
AND descricao ILIKE '%premium%'
```

3. JOIN

JOIN_SPECIALIST explica: "Combina dados de múltiplas tabelas"

```
-- INNER JOIN
SELECT p.nome, c.categoria
FROM produtos p
INNER JOIN categorias c ON p.categoria_id = c.id

-- LEFT JOIN
SELECT c.nome, COUNT(p.id) as total_pedidos
FROM clientes c
LEFT JOIN pedidos p ON c.id = p.cliente_id
GROUP BY c.nome

-- Multiple JOINS
SELECT p.nome, c.categoria, f.nome as fornecedor
FROM produtos p
JOIN categorias c ON p.categoria_id = c.id
JOIN fornecedores f ON p.fornecedor_id = f.id
```

4. GROUP BY

AGGREGATION_MASTER demonstra: "Agrupa resultados"

```
-- Agrupamento simples
SELECT categoria, COUNT(*) as total
FROM produtos
GROUP BY categoria

-- Múltiplas colunas
SELECT categoria, status, SUM(valor) as total
FROM vendas
GROUP BY categoria, status

-- Com HAVING
SELECT cliente_id, COUNT(*) as total_pedidos
FROM pedidos
GROUP BY cliente_id
HAVING COUNT(*) > 5
```

5. ORDER BY

SORT_EXPERT apresenta: "Ordena resultados"

```
-- Ordenação simples
ORDER BY data_criacao DESC

-- Múltiplas colunas
ORDER BY categoria ASC, preco DESC

-- Com expressões
ORDER BY (preco * quantidade) DESC
```

Funções e Expressões

1. Agregação

```
SELECT
    COUNT(*) as total,
    SUM(valor) as valor_total,
```

```
    AVG(preco) as preco_medio,
    MAX(data) as data_mais_recente,
    MIN(data) as data_mais_antiga
FROM vendas;
```

2. String

```
SELECT
    UPPER(nome) as nome_maiusculo,
    LOWER(email) as email_minusculo,
    SUBSTRING(descricao, 1, 100) as descricao_curta,
    CONCAT(nome, ' - ', categoria) as nome_completo
FROM produtos;
```

3. Data/Hora

```
SELECT
    DATE_TRUNC('month', data_criacao) as mes,
    EXTRACT(YEAR FROM data_criacao) as ano,
    data_criacao + INTERVAL '7 days' as data_vencimento
FROM pedidos;
```

Subconsultas

SUBQUERY_MASTER explica: "Consultas dentro de consultas"

```
-- Subconsulta no WHERE
SELECT nome
FROM produtos
WHERE categoria_id IN (
    SELECT id
    FROM categorias
    WHERE ativo = true
);
```

```
-- Subconsulta no FROM
```

```

SELECT dept_nome, total_funcionarios
FROM (
    SELECT departamento_id, COUNT(*) as total
    FROM funcionarios
    GROUP BY departamento_id
) f
JOIN departamentos d ON f.departamento_id = d.id;

-- Subconsulta correlacionada
SELECT p.nome
FROM produtos p
WHERE preco > (
    SELECT AVG(preco)
    FROM produtos
    WHERE categoria_id = p.categoria_id
);

```

Otimização

PERFORMANCE_GURU compartilha dicas essenciais:

1. Índices

```

-- Use índices apropriados
CREATE INDEX idx_produtos_categoria
ON produtos(categoria_id);

-- Índice composto para consultas frequentes
CREATE INDEX idx_pedidos_cliente_data
ON pedidos(cliente_id, data_criacao);

```

2. Análise de Execução

```

-- Analise o plano de execução
EXPLAIN ANALYZE
SELECT c.nome, COUNT(p.id) as total_pedidos
FROM clientes c

```

```
JOIN pedidos p ON c.id = p.cliente_id  
GROUP BY c.nome;
```

Boas Práticas

BEST_PRACTICE_MASTER recomenda:

1. Performance

- Evite SELECT *
- Use JOINs apropriados
- Limite resultados grandes
- Utilize índices corretamente

2. Legibilidade

- Use aliases descritivos
- Formate SQL adequadamente
- Comente consultas complexas
- Mantenha consistência

3. Manutenção

- Documente consultas importantes
- Use views para consultas comuns
- Implemente paginação
- Monitore performance

|| CHECKLIST SELECT: ||

- || Colunas necessárias apenas?
- || Índices apropriados?

- || JOINS otimizados? ||
- || WHERE eficiente? ||
- || Resultados limitados? ||

Conclusão

SELECT_SPECIALIST conclui: "O domínio do SELECT é fundamental para qualquer desenvolvedor SQL. Pratique diferentes tipos de consultas e sempre considere performance e manutenibilidade."



Dica Final: Teste suas consultas com volumes de dados realistas para garantir performance em produção.

INSERT: Inserindo Dados

```
|| DATA_MASTER >> Inserção de Dados  
|| INSTRUTOR: INSERT_SPECIALIST
```

Fundamentos do INSERT

INSERT_SPECIALIST explica: "O comando INSERT é fundamental para adicionar dados em suas tabelas."

Sintaxe Básica

```
-- Inserção simples  
INSERT INTO tabela (coluna1, coluna2)  
VALUES (valor1, valor2);  
  
-- Múltiplos registros  
INSERT INTO tabela (coluna1, coluna2)  
VALUES  
    (valor1, valor2),  
    (valor3, valor4),  
    (valor5, valor6);
```

Formas de Inserção

1. Valores Explícitos

```
-- Com todas as colunas  
INSERT INTO produtos (  
    nome,  
    preco,  
    categoria,  
    estoque  
) VALUES (
```

```

    'Smartphone X',
    999.99,
    'Eletrônicos',
    100
);

-- Com colunas específicas
INSERT INTO usuarios (
    email,
    nome
) VALUES (
    'user@email.com',
    'João Silva'
);

```

2. Inserção via SELECT

DATA_TRANSFER_EXPERT demonstra: "Copie dados de outras tabelas!"

```

-- Inserção básica via SELECT
INSERT INTO produtos_backup
SELECT * FROM produtos
WHERE categoria = 'Eletrônicos';

-- Inserção seletiva
INSERT INTO resumo_vendas (produto, total_vendido)
SELECT
    p.nome,
    SUM(v.quantidade)
FROM vendas v
JOIN produtos p ON v.produto_id = p.id
GROUP BY p.nome;

```

3. Valores Default e Expressões

```

-- Usando DEFAULT
INSERT INTO logs (

```

```

    evento,
    data_registro
) VALUES (
    'Login usuário',
    DEFAULT -- Usa CURRENT_TIMESTAMP
);

-- Com expressões
INSERT INTO pedidos (
    produto_id,
    quantidade,
    valor_total
) VALUES (
    101,
    5,
    (SELECT preco FROM produtos WHERE id = 101) * 5
);

```

Tratamento de Erros

ERROR_HANDLER apresenta: "Lidando com conflitos e erros"

```

-- Ignorar duplicados
INSERT INTO logs (id, mensagem)
ON CONFLICT (id) DO NOTHING;

-- Atualizar em caso de conflito
INSERT INTO produtos (codigo, nome, estoque)
VALUES ('P123', 'Novo Produto', 100)
ON CONFLICT (codigo)
DO UPDATE SET
    estoque = produtos.estoque + EXCLUDED.estoque;

```

Inserção em Massa

BULK_INSERT_MASTER compartilha: "Otimize inserções grandes!"

```
-- Preparando statement
PREPARE insert_produto (text, decimal, int) AS
INSERT INTO produtos (nome, preco, estoque)
VALUES ($1, $2, $3);

-- Executando
EXECUTE insert_produto('Produto A', 99.99, 50);
EXECUTE insert_produto('Produto B', 149.99, 30);

-- Limpando
DEALLOCATE insert_produto;
```

Boas Práticas

BEST_PRACTICES_SAGE recomenda:

1. Performance

- Use inserção em lote para múltiplos registros
- Considere desativar índices para cargas grandes
- Utilize transações apropriadamente
- Monitore o tamanho dos lotes

2. Integridade

- Valide dados antes da inserção
- Use constraints apropriadas
- Mantenha consistência referencial
- Trate valores nulos adequadamente

3. Manutenção

- Documente processos de carga
- Mantenha logs de inserção
- Implemente rollback strategy
- Monitore espaço em disco

|| CHECKLIST INSERT:

- || Dados validados?
- || Constraints verificadas?
- || Transação necessária?
- || Índices considerados?
- || Backup realizado?

Exemplos Práticos

Sistema de E-commerce

```
-- Inserindo novo produto
INSERT INTO produtos (
    codigo,
    nome,
    descricao,
    preco,
    estoque,
    categoria_id,
    criado_em,
    status
) VALUES (
    'PROD-123',
    'Smartphone Ultimate',
    'Smartphone de última geração',
    1299.99,
    50,
    (SELECT id FROM categorias WHERE nome = 'Eletrônicos'),
```

```

CURRENT_TIMESTAMP,
'ativo'
);

-- Registrando pedido
INSERT INTO pedidos (
    cliente_id,
    valor_total,
    status,
    data_pedido
) VALUES (
    1001,
    1299.99,
    'pendente',
    CURRENT_TIMESTAMP
);

-- Inserindo itens do pedido
INSERT INTO itens_pedido (
    pedido_id,
    produto_id,
    quantidade,
    preco_unitario
) VALUES (
    CURRVAL('pedidos_id_seq'),
    (SELECT id FROM produtos WHERE codigo = 'PROD-123'),
    1,
    1299.99
);

```

Troubleshooting Comum

PROBLEM_SOLVER apresenta soluções para problemas frequentes:

1. Violção de Chave Única

```
-- Verificando existência
SELECT COUNT(*)
FROM produtos
WHERE codigo = 'PROD-123';

-- Inserção segura
INSERT INTO produtos (codigo, nome)
SELECT 'PROD-123', 'Novo Produto'
WHERE NOT EXISTS (
    SELECT 1
    FROM produtos
    WHERE codigo = 'PROD-123'
);
```

2. Violação de Chave Estrangeira

```
-- Verificando referência
SELECT id
FROM categorias
WHERE id = 5;

-- Inserção com verificação
INSERT INTO produtos (categoria_id, nome)
SELECT 5, 'Novo Produto'
WHERE EXISTS (
    SELECT 1
    FROM categorias
    WHERE id = 5
);
```

Conclusão

INSERT_SPECIALIST conclui: "O domínio do INSERT vai além da sintaxe básica. Considere sempre performance, integridade e manutenibilidade em suas operações de inserção."



Dica Final: Sempre teste suas inserções em ambiente de desenvolvimento antes de aplicar em produção.

UPDATE: Modificando Dados

|| DATA_MASTER >> Atualização de Dados
|| INSTRUTOR: UPDATE_SPECIALIST

Fundamentos do UPDATE

UPDATE_SPECIALIST explica: "O comando UPDATE permite modificar dados existentes de forma precisa e controlada."

Sintaxe Básica

```
-- Atualização simples
UPDATE tabela
SET coluna = valor
WHERE condição;
```

```
-- Múltiplas colunas
UPDATE tabela
SET
    coluna1 = valor1,
    coluna2 = valor2
WHERE condição;
```

Formas de Atualização

1. Valores Diretos

```
-- Atualização simples
UPDATE produtos
SET
    preco = 999.99,
    ultima_atualizacao = CURRENT_TIMESTAMP
WHERE id = 1001;
```

```
-- Com expressões
UPDATE produtos
SET preco = preco * 1.1
WHERE categoria = 'Eletrônicos';
```

2. Atualização com JOIN

DATA_SYNC_EXPERT demonstra: "Atualize baseado em outras tabelas!"

```
-- Atualização com JOIN
UPDATE pedidos p
SET status = 'aprovado'
FROM clientes c
WHERE p.cliente_id = c.id
AND c.categoria = 'VIP';

-- Atualização múltipla
UPDATE produtos p
SET
    preco = p.preco * t.fator_ajuste,
    atualizado_em = CURRENT_TIMESTAMP
FROM tabela_ajustes t
WHERE p.categoria = t.categoria;
```

3. Atualizações Condicionais

```
-- Usando CASE
UPDATE funcionarios
SET salario = CASE
    WHEN tempo_servico < 2 THEN salario * 1.05
    WHEN tempo_servico < 5 THEN salario * 1.07
    ELSE salario * 1.10
END;

-- Com COALESCE
UPDATE clientes
```

```

SET
    telefone = COALESCE(novo_telefone, telefone),
    email = COALESCE(novo_email, email)
FROM atualizacoes_clientes
WHERE clientes.id = atualizacoes_clientes.cliente_id;

```

Tratamento de Erros

ERROR_HANDLER apresenta: "Lidando com atualizações seguras"

```

-- Com validação
UPDATE produtos
SET preco = novo_preco
WHERE id = produto_id
AND novo_preco > 0
RETURNING id, nome, preco as novo_preco;

-- Dentro de transação
BEGIN;
    UPDATE contas
    SET saldo = saldo - 100
    WHERE id = 1;

    UPDATE contas
    SET saldo = saldo + 100
    WHERE id = 2;

-- Verificação
IF EXISTS (SELECT 1 FROM contas WHERE saldo < 0) THEN
    ROLLBACK;
ELSE
    COMMIT;
END IF;

```

Atualizações em Massa

BULK_UPDATE_MASTER compartilha: "Otimize atualizações grandes!"

```
-- Atualização em lotes
UPDATE produtos
SET status = 'inativo'
WHERE id IN (
    SELECT id
    FROM produtos
    WHERE ultima_venda < CURRENT_DATE - INTERVAL '1 year'
    LIMIT 1000
);

-- Com tabela temporária
CREATE TEMP TABLE updates_temp AS
SELECT id, novo_preco
FROM calculos_precos;

UPDATE produtos p
SET preco = t.novo_preco
FROM updates_temp t
WHERE p.id = t.id;
```

Boas Práticas

BEST_PRACTICES_SAGE recomenda:

1. Performance

- Use índices adequadamente
- Atualize em lotes quando possível
- Evite triggers desnecessários
- Monitore locks e deadlocks

2. Integridade

- Valide dados antes da atualização
- Use transações quando necessário
- Mantenha consistência referencial
- Considere impacto em constraints

3. Manutenção

- Mantenha logs de alterações
- Documente atualizações críticas
- Implemente rollback strategy
- Monitore impacto em índices

CHECKLIST UPDATE:	
<input type="checkbox"/> WHERE adequado?	
<input type="checkbox"/> Dados validados?	
<input type="checkbox"/> Transação necessária?	
<input type="checkbox"/> Índices considerados?	
<input type="checkbox"/> Backup realizado?	

Exemplos Práticos

Sistema de E-commerce

```
-- Atualização de estoque após venda
UPDATE produtos p
SET
    estoque = p.estoque - v.quantidade,
    ultima_venda = CURRENT_TIMESTAMP,
    total_vendas = p.total_vendas + 1
FROM vendas v
WHERE p.id = v.producto_id
```

```

AND v.status = 'confirmado';

-- Atualização de status de pedidos
UPDATE pedidos p
SET
    status = 'em_transito',
    atualizado_em = CURRENT_TIMESTAMP,
    tracking_code = e.codigo_rastreio
FROM entregas e
WHERE p.id = e.pedido_id
AND p.status = 'aprovado'
AND e.status = 'coletado';

```

Troubleshooting Comum

PROBLEM_SOLVER apresenta soluções para problemas frequentes:

1. Deadlocks em Atualizações

```

-- Identificando locks
SELECT relation::regclass, mode, granted
FROM pg_locks
WHERE NOT granted;

-- Atualizando com timeout
SET lock_timeout = '10s';
UPDATE tabela_concorrida
SET status = 'processado'
WHERE id = 1001;

```

2. Atualizações Órfãs

```

-- Verificando integridade
UPDATE pedidos p
SET status = 'cancelado'

```

```
WHERE EXISTS (
    SELECT 1
    FROM clientes c
    WHERE p.cliente_id = c.id
    AND c.status = 'inativo'
);
```

Conclusão

UPDATE_SPECIALIST conclui: "O UPDATE é uma operação poderosa que requer atenção aos detalhes. Sempre valide suas condições e considere o impacto das alterações."



Dica Final: Sempre teste suas atualizações com SELECT primeiro para confirmar o conjunto de registros afetados.

DELETE: Removendo Dados com Precisão

```
|| DATA_MASTER >> Remoção de Dados  
|| INSTRUTOR: DELETE_SPECIALIST
```

Fundamentos do DELETE

DELETE_SPECIALIST explica: "O comando DELETE permite remover registros de forma precisa e controlada."

Sintaxe Básica

```
-- Remoção simples  
DELETE FROM tabela  
WHERE condição;  
  
-- Remoção com RETURNING  
DELETE FROM produtos  
WHERE estoque = 0  
RETURNING id, nome;
```

Formas de Deleção

1. Deleção Simples

```
-- Remoção por ID  
DELETE FROM pedidos  
WHERE id = 1001;  
  
-- Remoção por condição
```

```
DELETE FROM produtos
WHERE validade < CURRENT_DATE;
```

2. Deleção com JOIN

DATA_SYNC_EXPERT demonstra: "Delete registros baseado em outras tabelas!"

```
-- Deleção com JOIN
DELETE FROM pedidos p
USING clientes c
WHERE p.cliente_id = c.id
AND c.status = 'inativo';

-- Deleção múltipla
DELETE FROM produtos
WHERE categoria_id IN (
    SELECT id
    FROM categorias
    WHERE descontinuada = true
);
```

3. Deleção Condicional

```
-- Usando subquery
DELETE FROM logs
WHERE created_at < (
    SELECT CURRENT_TIMESTAMP - INTERVAL '30 days'
);

-- Com EXISTS
DELETE FROM carrinho
WHERE EXISTS (
    SELECT 1
    FROM produtos p
    WHERE carrinho.producto_id = p.id
);
```

```
    AND p.descontinuado = true  
);
```

Deleção em Massa

BULK_DELETE_MASTER compartilha: "Otimize deleções grandes!"

```
-- Deleção em lotes  
DO $$  
DECLARE  
    batch_size INTEGER := 1000;  
BEGIN  
    WHILE EXISTS (  
        SELECT 1  
        FROM logs  
        WHERE created_at < CURRENT_DATE - INTERVAL '1 year'  
    ) LOOP  
        DELETE FROM logs  
        WHERE id IN (  
            SELECT id  
            FROM logs  
            WHERE created_at < CURRENT_DATE - INTERVAL '1 year'  
            LIMIT batch_size  
        );  
        COMMIT;  
    END LOOP;  
END $$;
```

Soft Delete

ARCHITECTURE_SAGE apresenta: "Às vezes, é melhor não deletar de verdade!"

```
-- Implementando soft delete  
ALTER TABLE produtos  
ADD COLUMN deleted_at TIMESTAMP,  
ADD COLUMN is_active BOOLEAN DEFAULT true;
```

```
-- Realizando soft delete
UPDATE produtos
SET
    deleted_at = CURRENT_TIMESTAMP,
    is_active = false
WHERE id = 1001;

-- Consultando apenas ativos
SELECT * FROM produtos
WHERE is_active = true
OR is_active IS NULL;
```

Boas Práticas

BEST_PRACTICES_SAGE recomenda:

1. Performance

- Use índices adequadamente
- Delete em lotes quando necessário
- Considere TRUNCATE para limpeza total
- Monitore locks e deadlocks

2. Integridade

- Verifique constraints
- Use transações quando necessário
- Considere impacto em dados relacionados
- Mantenha consistência referencial

3. Segurança

- Faça backup antes de deleções grandes
- Valide condições WHERE cuidadosamente
- Implemente soft delete quando apropriado
- Mantenha logs de deleções importantes

|| CHECKLIST DELETE:

- || WHERE adequado?
- || Impacto analisado?
- || Transação necessária?
- || Backup realizado?
- || Soft delete considerado?

Exemplos Práticos

Sistema de E-commerce

```
-- Limpeza de carrinhos abandonados
DELETE FROM carrinhos
WHERE ultima_atualizacao < CURRENT_TIMESTAMP - INTERVAL '24 hours'
AND status = 'pendente';

-- Remoção de produtos descontinuados
DELETE FROM produtos p
WHERE NOT EXISTS (
    SELECT 1
    FROM pedidos_itens pi
    WHERE pi.produto_id = p.id
    AND pi.created_at > CURRENT_DATE - INTERVAL '6 months'
)
AND p.estoque = 0;
```

Troubleshooting Comum

PROBLEM_SOLVER apresenta soluções para problemas frequentes:

1. Violação de Constraint

```
-- Verificando dependências
SELECT
    tc.table_schema,
    tc.table_name,
    kcu.column_name
FROM information_schema.table_constraints tc
JOIN information_schema.key_column_usage kcu
    ON tc.constraint_name = kcu.constraint_name
WHERE tc.constraint_type = 'FOREIGN KEY'
AND kcu.referenced_table_name = 'sua_tabela';

-- Deleção segura
BEGIN;
    DELETE FROM tabela_dependente
    WHERE chave_estrangeira IN (
        SELECT id FROM tabela_principal WHERE status = 'inativo'
    );

    DELETE FROM tabela_principal
    WHERE status = 'inativo';
COMMIT;
```

2. Recuperação de Dados

```
-- Backup antes de deletar
CREATE TABLE deleted_records AS
SELECT * FROM tabela
WHERE status = 'para_deletar';

-- Então execute a deleção
```

```
DELETE FROM tabela  
WHERE status = 'para_deletar';
```

Conclusão

DELETE_SPECIALIST conclui: "O DELETE é uma operação irreversível - use com cautela e sempre tenha um plano de backup."



Dica Final: Sempre teste suas condições de DELETE com um SELECT primeiro para confirmar exatamente quais registros serão afetados.

MERGE: Sincronizando Dados com Precisão

|| DATA_SYNC_MASTER >> Sincronização de Dados
|| INSTRUTOR: MERGE_SPECIALIST

Fundamentos do MERGE

MERGE_SPECIALIST explica: "O comando MERGE é uma ferramenta poderosa para sincronização de dados, combinando INSERT, UPDATE e DELETE em uma única operação."

Sintaxe Básica

```
MERGE INTO tabela_destino d
USING tabela_origem o
ON (d.id = o.id)
WHEN MATCHED THEN
    UPDATE SET coluna = o.valor
WHEN NOT MATCHED THEN
    INSERT (coluna) VALUES (o.valor);
```

Padrões de Uso

1. Sincronização Completa

```
-- Sincronização com todas as ações
MERGE INTO produtos_destino d
USING produtos_origem o
ON (d.codigo = o.codigo)
WHEN MATCHED THEN
    UPDATE SET
        nome = o.nome,
```

```

    preco = o.preco,
    updated_at = CURRENT_TIMESTAMP
WHEN NOT MATCHED THEN
    INSERT (codigo, nome, preco, created_at)
    VALUES (o.codigo, o.nome, o.preco, CURRENT_TIMESTAMP)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE;

```

2. Atualização Condisional

```

-- Merge com condições específicas
MERGE INTO estoque e
USING novos_dados n
ON (e.producto_id = n.producto_id)
WHEN MATCHED AND e.quantidade != n.quantidade THEN
    UPDATE SET
        quantidade = n.quantidade,
        ultima_atualizacao = CURRENT_TIMESTAMP
WHEN NOT MATCHED AND n.quantidade > 0 THEN
    INSERT (produto_id, quantidade, ultima_atualizacao)
    VALUES (n.producto_id, n.quantidade, CURRENT_TIMESTAMP);

```

Casos de Uso

DATA_SYNC_EXPERT apresenta: "Cenários comuns para MERGE"

1. Dimensões Slowly Changing

```

-- Dimensão Tipo 2
MERGE INTO dim_clientes d
USING stg_clientes s
ON (d.cliente_id = s.cliente_id AND d.atual = true)
WHEN MATCHED AND (
    d.nome != s.nome OR
    d.endereco != s.endereco
) THEN
    UPDATE SET

```

```

        atual = false,
        data_fim = CURRENT_TIMESTAMP
WHEN NOT MATCHED THEN
    INSERT (cliente_id, nome, endereco, data_inicio, data_fim,
atual)
    VALUES (s.cliente_id, s.nome, s.endereco, CURRENT_TIMESTAMP,
NULL, true);

```

2. Sincronização de Cache

```

-- Atualização de cache
MERGE INTO cache_produtos c
USING produtos_atualizados p
ON (c.producto_id = p.id)
WHEN MATCHED AND c.hash != p.hash THEN
    UPDATE SET
        dados = p.dados,
        hash = p.hash,
        atualizado_em = CURRENT_TIMESTAMP
WHEN NOT MATCHED THEN
    INSERT (produto_id, dados, hash, atualizado_em)
    VALUES (p.id, p.dados, p.hash, CURRENT_TIMESTAMP);

```

Boas Práticas

BEST_PRACTICES_SAGE compartilha:

1. Performance

- Use índices apropriados
- Considere o volume de dados
- Faça em lotes quando necessário
- Monitore tempos de execução

2. Integridade

- Use transações
- Valide dados antes do merge
- Mantenha consistência
- Implemente logging

3. Manutenção

- Documente regras de merge
- Mantenha histórico de execuções
- Implemente recuperação de erros
- Monitore resultados

CHECKLIST MERGE:

- Dados validados?
- Índices verificados?
- Transação configurada?
- Logging implementado?
- Recuperação planejada?

Padrões Avançados

PATTERN_MASTER demonstra: "Técnicas avançadas de MERGE"

1. Merge com Staging

```
-- Usando tabela temporária
WITH staging AS (
    SELECT * FROM novos_dados
    WHERE data_processamento = CURRENT_DATE
```

```

)
MERGE INTO produtos_final p
USING staging s
ON (p.id = s.id)
WHEN MATCHED THEN
    UPDATE SET
        dados = s.dados,
        atualizado_em = CURRENT_TIMESTAMP
WHEN NOT MATCHED THEN
    INSERT (id, dados, criado_em)
    VALUES (s.id, s.dados, CURRENT_TIMESTAMP);

```

2. Merge com Auditoria

```

-- Mantendo histórico de alterações
MERGE INTO produtos p
USING atualizacoes a
ON (p.id = a.id)
WHEN MATCHED THEN
    UPDATE SET
        nome = a.nome,
        preco = a.preco
        -- Registro de auditoria
        INSERT INTO produtos_audit (
            produto_id, campo, valor_antigo, valor_novo,
            data_alteracao
        )
        VALUES
            (p.id, 'nome', p.nome, a.nome, CURRENT_TIMESTAMP),
            (p.id, 'preco', p.preco, a.preco, CURRENT_TIMESTAMP)
WHEN NOT MATCHED THEN
    INSERT (id, nome, preco)
    VALUES (a.id, a.nome, a.preco);

```

Troubleshooting

ERROR_HANDLER apresenta soluções para problemas comuns:

1. Conflitos de Chave

```
-- Verificação prévia
SELECT COUNT(*), id
FROM tabela_origem
GROUP BY id
HAVING COUNT(*) > 1;

-- Merge com resolução de duplicatas
MERGE INTO destino d
USING (
    SELECT DISTINCT ON (id) *
    FROM origem
    ORDER BY id, ultima_atualizacao DESC
) o
ON (d.id = o.id)
...
```

2. Performance

```
-- Merge em lotes
DO $$

DECLARE
    batch_size INTEGER := 1000;
    total_rows INTEGER;
BEGIN
    SELECT COUNT(*) INTO total_rows FROM origem;

    FOR i IN 0..total_rows/batch_size LOOP
        MERGE INTO destino d
        USING (
            SELECT *
            FROM origem
            OFFSET i * batch_size
```

```
    LIMIT batch_size
) o
ON (d.id = o.id)
. . . ;
COMMIT;
END LOOP;
END $$;
```

Conclusão

MERGE_SPECIALIST conclui: "O MERGE é uma ferramenta poderosa para sincronização de dados, mas requer planejamento cuidadoso e atenção aos detalhes."



Dica Final: Sempre teste seu MERGE com um conjunto pequeno de dados antes de executar em produção, e mantenha backups atualizados.

Exercícios de DML (Data Manipulation Language)

Visão Geral

Este módulo contém uma série progressiva de exercícios para praticar comandos DML em SQL. Os exercícios estão organizados em três níveis de dificuldade:

Nível Básico

- Consultas simples com SELECT
- Inserções básicas de dados
- Atualizações e exclusões simples
- Operações fundamentais de DML

Nível Intermediário

- Consultas com múltiplas tabelas
- Subconsultas e expressões
- Operações em lote
- Técnicas de manipulação avançadas

Nível Avançado

- Operações complexas de sincronização
- Manipulação de dados em grande escala
- Otimização de performance
- Cenários empresariais realistas

Estrutura dos Exercícios

Cada exercício segue o formato:

1. Descrição do problema
2. Requisitos específicos
3. Dicas de implementação
4. Solução de referência
5. Critérios de avaliação

Ambiente de Prática

```
-- Database de teste
CREATE DATABASE exercicios_dml;

-- Schema para isolamento
CREATE SCHEMA pratica;

-- Tabelas de exemplo
CREATE TABLE pratica.clientes (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    data_cadastro DATE DEFAULT CURRENT_DATE
);

CREATE TABLE pratica.produtos (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    preco DECIMAL(10,2) CHECK (preco > 0),
    estoque INTEGER DEFAULT 0
);

CREATE TABLE pratica_pedidos (
```

```

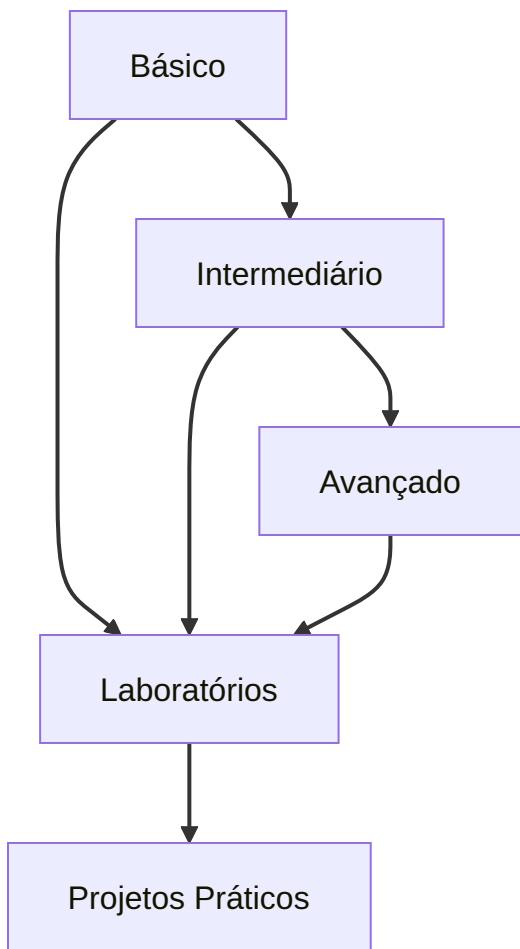
        id SERIAL PRIMARY KEY,
        cliente_id INTEGER REFERENCES pratica.clientes(id),
        data_pedido TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        status VARCHAR(20) DEFAULT 'Pendente'
);

CREATE TABLE pratica.itens_pedido (
    pedido_id INTEGER REFERENCES pratica.pedidos(id),
    produto_id INTEGER REFERENCES pratica.produtos(id),
    quantidade INTEGER CHECK (quantidade > 0),
    preco_unitario DECIMAL(10, 2),
    PRIMARY KEY (pedido_id, produto_id)
);

-- Tabela de controle
CREATE TABLE pratica.controle_exercicios (
    id SERIAL PRIMARY KEY,
    exercicio VARCHAR(50),
    completado BOOLEAN DEFAULT FALSE,
    data_conclusao TIMESTAMP
);

```

Fluxo de Estudo Recomendado



Preparação do Ambiente

Para obter o máximo proveito destes exercícios, recomendamos:

1. Configurar um ambiente de banco de dados

- PostgreSQL 12+ ou MySQL 8+
- Cliente SQL (DBeaver, pgAdmin, MySQL Workbench)
- Scripts de inicialização fornecidos

2. Dados de teste

```

-- Inserir dados de exemplo
INSERT INTO pratica.clientes (nome, email)
VALUES
  ('Ana Silva', 'ana@email.com'),
  ('João Souza', 'joao@email.com'),
  ('Maria Oliveira', 'maria@email.com'),
  ('Pedro Gomes', 'pedro@email.com'),
  ('Raquel Ferreira', 'raquel@email.com')
  
```

```

('Bruno Costa', 'bruno@email.com'),
('Carla Mendes', 'carla@email.com');

INSERT INTO pratica.produtos (nome, preco, estoque)
VALUES
  ('Smartphone', 1299.99, 50),
  ('Notebook', 3499.99, 20),
  ('Fones de Ouvido', 199.99, 100);

```

3. Ferramentas recomendadas

- Editor de SQL com highlight de sintaxe
- Ferramenta de visualização de planos de execução
- Ambiente para documentar suas soluções

Navegação do Conteúdo

- Exercícios Básicos ([Exercícios Básicos de DML](#))
- Exercícios Intermediários ([Exercícios Intermediários de DML](#))
- Exercícios Avançados ([Exercícios Avançados de DML](#))

Dicas de Estudo

BOAS PRÁTICAS:

- ✓ Teste cada comando individualmente
- ✓ Analise planos de execução
- ✓ Documente suas soluções
- ✓ Pratique regularmente
- ✓ Revise conceitos quando necessário

Supporte e Recursos

- Fórum de discussão
- Documentação oficial do SGBD
- Exemplos práticos adicionais
- Soluções comentadas

Próximos Passos

Após completar os exercícios de DML, recomendamos:

1. Laboratórios práticos

- Aplicação em cenários realistas
- Integração com outros conceitos

2. Projetos integrados

- Desenvolvimento de aplicações completas
- Implementação de sistemas de banco de dados

3. Tópicos avançados

- Otimização de consultas
- Tuning de performance
- Padrões de design SQL

Feedback e Avaliação

Para cada conjunto de exercícios, você pode:

- Verificar suas soluções contra as referências
- Compartilhar abordagens alternativas

- Discutir otimizações possíveis
- Solicitar revisão por pares



Nota: Estes exercícios são projetados para serem progressivos.
Recomendamos completar os exercícios básicos antes de avançar para os níveis mais complexos.

Exercícios Básicos de DML

Exercício 1: Consultas Simples

Descrição

Pratique consultas básicas para recuperar dados das tabelas.

Requisitos

- Selecionar todos os registros
- Selecionar colunas específicas
- Filtrar com WHERE
- Ordenar resultados

Solução

```
-- Selecionar todos os registros
SELECT * FROM pratica.clientes;

-- Selecionar colunas específicas
SELECT nome, email FROM pratica.produtos;

-- Filtrar com WHERE
SELECT * FROM pratica.produtos
WHERE preco < 1000;

-- Ordenar resultados
SELECT * FROM pratica.clientes
ORDER BY nome ASC;
```

Exercício 2: Inserção de Dados

Descrição

Pratique a inserção de novos registros nas tabelas.

Requisitos

- Inserir um único registro
- Inserir múltiplos registros
- Inserir com valores padrão
- Inserir com subconsulta

Solução

```
-- Inserir um único registro
INSERT INTO pratica.clientes (nome, email)
VALUES ('Daniel Santos', 'daniel@email.com');

-- Inserir múltiplos registros
INSERT INTO pratica.produtos (nome, preco, estoque)
VALUES
  ('Tablet', 899.99, 30),
  ('Mouse', 49.99, 100);

-- Inserir com valores padrão
INSERT INTO pratica.clientes (nome)
VALUES ('Eduardo Lima');

-- Inserir com subconsulta
INSERT INTO pratica_pedidos (cliente_id, status)
SELECT id, 'Novo'
FROM pratica.clientes
WHERE nome = 'Ana Silva';
```

Exercício 3: Atualização de Dados

Descrição

Pratique a atualização de registros existentes.

Requisitos

- Atualizar um único registro
- Atualizar múltiplos registros
- Atualizar com condição
- Atualizar com expressão

Solução

```
-- Atualizar um único registro
UPDATE pratica.clientes
SET email = 'ana.silva@email.com'
WHERE id = 1;

-- Atualizar múltiplos registros
UPDATE pratica.produtos
SET estoque = estoque + 10
WHERE estoque < 30;

-- Atualizar com condição
UPDATE pratica.pedidos
SET status = 'Aprovado'
WHERE cliente_id IN (
    SELECT id FROM pratica.clientes
    WHERE nome LIKE 'Ana%'
);

-- Atualizar com expressão
UPDATE pratica.produtos
SET preco = preco * 1.1
WHERE nome = 'Smartphone';
```

Exercício 4: Exclusão de Dados

Descrição

Pratique a exclusão de registros das tabelas.

Requisitos

- Excluir um único registro
- Excluir múltiplos registros
- Excluir com condição
- Excluir com subconsulta

Solução

```
-- Excluir um único registro
DELETE FROM pratica.clientes
WHERE id = 5;

-- Excluir múltiplos registros
DELETE FROM pratica.produtos
WHERE estoque = 0;

-- Excluir com condição
DELETE FROM pratica_pedidos
WHERE status = 'Cancelado'
AND data_pedido < CURRENT_DATE - INTERVAL '30 days';

-- Excluir com subconsulta
DELETE FROM pratica_itens_pedido
WHERE produto_id IN (
    SELECT id FROM pratica_produtos
    WHERE preco > 3000
);
```

Exercício 5: Consultas com Funções

Descrição

Pratique consultas utilizando funções SQL básicas.

Requisitos

- Funções de agregação
- Funções de string
- Funções de data
- Funções matemáticas

Solução

```
-- Funções de agregação
SELECT
    COUNT(*) as total_clientes,
    MAX(data_cadastro) as cadastro_mais_recente
FROM pratica.clientes;

-- Funções de string
SELECT
    UPPER(nome) as nome_maiusculo,
    LENGTH(email) as tamanho_email
FROM pratica.clientes;

-- Funções de data
SELECT
    id,
    data_pedido,
    EXTRACT(MONTH FROM data_pedido) as mes,
    EXTRACT(YEAR FROM data_pedido) as ano
FROM pratica.pedidos;

-- Funções matemáticas
```

```
SELECT
    nome,
    preco,
    ROUND(preco * 0.9, 2) as preco_com_desconto
FROM pratica.produtos;
```

Exercício 6: Consultas com GROUP BY

Descrição

Pratique consultas com agrupamento de dados.

Requisitos

- Agrupar por uma coluna
- Agrupar por múltiplas colunas
- Filtrar grupos com HAVING
- Ordenar grupos

Solução

```
-- Agrupar por uma coluna
SELECT
    status,
    COUNT(*) as total_pedidos
FROM pratica.pedidos
GROUP BY status;

-- Agrupar por múltiplas colunas
SELECT
    EXTRACT(YEAR FROM data_pedido) as ano,
    EXTRACT(MONTH FROM data_pedido) as mes,
    status,
    COUNT(*) as total_pedidos
FROM pratica.pedidos
```

```
GROUP BY ano, mes, status;

-- Filtrar grupos com HAVING
SELECT
    cliente_id,
    COUNT(*) as total_pedidos
FROM practica.pedidos
GROUP BY cliente_id
HAVING COUNT(*) > 3;

-- Ordenar grupos
SELECT
    status,
    COUNT(*) as total_pedidos
FROM practica.pedidos
GROUP BY status
ORDER BY total_pedidos DESC;
```

Exercício 7: Consultas com JOIN

Descrição

Pratique consultas que relacionam múltiplas tabelas.

Requisitos

- INNER JOIN
- LEFT JOIN
- Múltiplos JOINs
- Filtros com JOIN

Solução

```
-- INNER JOIN
SELECT
```

```

c.nome as cliente,
p.id as pedido_id,
p.data_pedido
FROM practica.clientes c
INNER JOIN practica.pedidos p ON c.id = p.cliente_id;

-- LEFT JOIN
SELECT
    c.nome as cliente,
    COUNT(p.id) as total_pedidos
FROM practica.clientes c
LEFT JOIN practica.pedidos p ON c.id = p.cliente_id
GROUP BY c.nome;

-- Múltiplos JOINS
SELECT
    c.nome as cliente,
    p.id as pedido_id,
    pr.nome as produto,
    ip.quantidade
FROM practica.clientes c
JOIN practica.pedidos p ON c.id = p.cliente_id
JOIN practica.itens_pedido ip ON p.id = ip.pedido_id
JOIN practica.produtos pr ON ip.producto_id = pr.id;

-- Filtros com JOIN
SELECT
    c.nome as cliente,
    p.id as pedido_id
FROM practica.clientes c
JOIN practica.pedidos p ON c.id = p.cliente_id
WHERE p.status = 'Aprovado'
AND p.data_pedido > CURRENT_DATE - INTERVAL '7 days';

```

Exercício 8: Subconsultas

Descrição

Pratique o uso de subconsultas em operações DML.

Requisitos

- Subconsulta no WHERE
- Subconsulta no FROM
- Subconsulta no SELECT
- Subconsulta com operadores de comparação

Solução

```
-- Subconsulta no WHERE
SELECT * FROM pratica.clientes
WHERE id IN (
    SELECT DISTINCT cliente_id
    FROM pratica_pedidos
    WHERE status = 'Aprovado'
);

-- Subconsulta no FROM
SELECT
    cliente,
    total_pedidos
FROM (
    SELECT
        c.nome as cliente,
        COUNT(p.id) as total_pedidos
    FROM pratica.clientes c
    LEFT JOIN pratica_pedidos p ON c.id = p.cliente_id
    GROUP BY c.nome
) as resumo
WHERE total_pedidos > 0;

-- Subconsulta no SELECT
```

```

SELECT
    p.id,
    p.nome,
    p.preco,
    (SELECT AVG(preco) FROM pratica.produtos) as preco_medio,
    p.preco - (SELECT AVG(preco) FROM pratica.produtos) as
diferenca
FROM pratica.produtos p;

-- Subconsulta com operadores de comparação
SELECT * FROM pratica.produtos
WHERE preco > (
    SELECT AVG(preco) FROM pratica.produtos
);

```

Exercício 9: Operadores de Conjunto

Descrição

Pratique o uso de operadores de conjunto em consultas.

Requisitos

- UNION
- INTERSECT
- EXCEPT
- Combinação de operadores

Solução

```

-- UNION
SELECT nome, email, 'Cliente' as tipo
FROM pratica.clientes
UNION
SELECT nome, 'N/A', 'Produto' as tipo

```

```

FROM pratica.produtos;

-- INTERSECT
SELECT cliente_id
FROM pratica.pedidos
WHERE status = 'Aprovado'
INTERSECT
SELECT cliente_id
FROM pratica.pedidos
WHERE status = 'Entregue';

-- EXCEPT
SELECT id FROM pratica.clientes
EXCEPT
SELECT cliente_id FROM pratica.pedidos;

-- Combinação de operadores
(SELECT cliente_id FROM pratica.pedidos WHERE status = 'Aprovado')
UNION
(SELECT cliente_id FROM pratica.pedidos WHERE status = 'Entregue')
EXCEPT
(SELECT cliente_id FROM pratica.pedidos WHERE status =
'Cancelado');

```

Exercício 10: Transações

Descrição

Pratique o uso de transações para garantir a integridade dos dados.

Requisitos

- Iniciar transação
- Confirmar alterações
- Reverter alterações

- Pontos de salvamento

Solução

```
-- Iniciar transação
BEGIN;

-- Operações dentro da transação
INSERT INTO pratica.clientes (nome, email)
VALUES ('Fernando Gomes', 'fernando@email.com');

UPDATE pratica.produtos
SET estoque = estoque - 5
WHERE nome = 'Smartphone';

-- Confirmar alterações
COMMIT;

-- Exemplo com rollback
BEGIN;

UPDATE pratica.produtos
SET preco = preco * 0.5;

-- Ops, desconto muito alto!
ROLLBACK;

-- Exemplo com savepoint
BEGIN;

INSERT INTO pratica.clientes (nome, email)
VALUES ('Gabriela Lima', 'gabriela@email.com');

SAVEPOINT novo_cliente;

UPDATE pratica.produtos
SET estoque = 0;
```

```
-- Ops, não queremos zerar o estoque!
ROLLBACK TO novo_cliente;

-- Continuar com outras operações
INSERT INTO pratica.pedidos (cliente_id, status)
VALUES ((SELECT id FROM pratica.clientes WHERE nome = 'Gabriela
Lima'), 'Novo');

COMMIT;
```

Critérios de Avaliação

- Sintaxe correta
- Uso adequado de cláusulas e operadores
- Eficiência das consultas
- Integridade dos dados após manipulação
- Aplicação de boas práticas

Dicas de Estudo

- Teste cada comando separadamente
- Verifique os resultados após cada operação
- Pratique com diferentes conjuntos de dados
- Experimente variações dos comandos
- Analise o plano de execução das consultas



Nota: Estes exercícios são fundamentais para construir uma base sólida em manipulação de dados. Certifique-se de compreender completamente cada

conceito antes de avançar para os exercícios intermediários.

Exercícios Intermediários de DML

Exercício 1: Operações em Lote

Descrição

Pratique operações DML em lote para manipular múltiplos registros de forma eficiente.

Requisitos

- Inserção em lote com SELECT
- Atualização em lote com JOIN
- Exclusão em lote com subconsulta
- Operações condicionais em lote

Solução

```
-- Inserção em lote com SELECT
INSERT INTO pratica.controle_exercicios (exercicio, completado)
SELECT 'Exercício ' || id, false
FROM generate_series(1, 10) as id;

-- Atualização em lote com JOIN
UPDATE pratica.produtos p
SET estoque = p.estoque - ip.quantidade
FROM pratica.itens_pedido ip
JOIN pratica_pedidos pe ON ip.pedido_id = pe.id
WHERE ip.producto_id = p.id
AND pe.status = 'Aprovado'
AND pe.data_pedido >= CURRENT_DATE - INTERVAL '1 day';

-- Exclusão em lote com subconsulta
DELETE FROM pratica.itens_pedido
WHERE pedido_id IN (
    SELECT id
```

```

    FROM pratica.pedidos
    WHERE status = 'Cancelado'
    AND data_pedido < CURRENT_DATE - INTERVAL '90 days'
);

-- Operações condicionais em lote
UPDATE pratica.produtos
SET
    preco = CASE
        WHEN estoque > 100 THEN preco * 0.9 -- Desconto para
produtos com muito estoque
        WHEN estoque < 10 THEN preco * 1.1 -- Aumento para
produtos com pouco estoque
        ELSE preco -- Mantém o preço
para estoque normal
    END,
    nome = CASE
        WHEN nome NOT LIKE '%Novo%' AND dataCadastro >
CURRENT_DATE - INTERVAL '30 days'
        THEN nome || ' (Novo)'
        ELSE nome
    END;

```

Exercício 2: Consultas Avançadas com JOIN

Descrição

Pratique consultas mais complexas utilizando diferentes tipos de JOIN.

Requisitos

- INNER JOIN com múltiplas tabelas
- LEFT/RIGHT JOIN com condições
- FULL OUTER JOIN
- CROSS JOIN

Solução

```
-- INNER JOIN com múltiplas tabelas
SELECT
    c.nome as cliente,
    p.id as pedido_id,
    p.data_pedido,
    SUM(ip.quantidade * ip.preco_unitario) as valor_total
FROM practica.clientes c
JOIN practica.pedidos p ON c.id = p.cliente_id
JOIN practica.itens_pedido ip ON p.id = ip.pedido_id
GROUP BY c.nome, p.id, p.data_pedido
ORDER BY p.data_pedido DESC;

-- LEFT/RIGHT JOIN com condições
SELECT
    c.nome as cliente,
    COUNT(p.id) as total_pedidos,
    COALESCE(SUM(ip.quantidade * ip.preco_unitario), 0) as
valor_total_compras
FROM practica.clientes c
LEFT JOIN practica.pedidos p ON c.id = p.cliente_id AND p.status != 'Cancelado'
LEFT JOIN practica.itens_pedido ip ON p.id = ip.pedido_id
GROUP BY c.nome
ORDER BY valor_total_compras DESC;

-- FULL OUTER JOIN
SELECT
    c.nome as cliente,
    p.id as pedido_id,
    p.status
FROM practica.clientes c
FULL OUTER JOIN practica.pedidos p ON c.id = p.cliente_id
WHERE c.id IS NULL OR p.id IS NULL;

-- CROSS JOIN
```

```

SELECT
    c.nome as cliente,
    pr.nome as produto,
    'Potencial interesse' as status
FROM practica.clientes c
CROSS JOIN practica.produtos pr
WHERE pr.categoria = 'Eletrônicos'
AND NOT EXISTS (
    SELECT 1 FROM practica_pedidos p
    JOIN practica_itens_pedido ip ON p.id = ip.pedido_id
    WHERE p.cliente_id = c.id
    AND ip.produto_id = pr.id
);

```

Exercício 3: Subconsultas Correlacionadas

Descrição

Pratique o uso de subconsultas correlacionadas para resolver problemas complexos.

Requisitos

- Subconsulta correlacionada no WHERE
- Subconsulta correlacionada no SELECT
- Subconsulta com EXISTS/NOT EXISTS
- Subconsulta com operadores de comparação

Solução

```

-- Subconsulta correlacionada no WHERE
SELECT c.nome, c.email
FROM practica.clientes c
WHERE (
    SELECT COUNT(*)
    FROM practica_pedidos p

```

```

        WHERE p.cliente_id = c.id
) > 3;

-- Subconsulta correlacionada no SELECT
SELECT
    p.nome as produto,
    p.preco,
    (
        SELECT AVG(ip.preco_unitario)
        FROM pratica.itens_pedido ip
        WHERE ip.producto_id = p.id
    ) as preco_medio_vendido
FROM pratica.produtos p;

-- Subconsulta com EXISTS/NOT EXISTS
SELECT c.nome
FROM pratica.clientes c
WHERE EXISTS (
    SELECT 1
    FROM pratica_pedidos p
    JOIN pratica_itens_pedido ip ON p.id = ip_pedido_id
    JOIN pratica_produtos pr ON ip.producto_id = pr.id
    WHERE p.cliente_id = c.id
    AND pr.preco > 1000
);

-- Subconsulta com operadores de comparação
SELECT p.nome, p.preco
FROM pratica.produtos p
WHERE p.preco > ALL (
    SELECT AVG(preco)
    FROM pratica.produtos
    GROUP BY categoria
);

```

Exercício 4: Expressões de Tabela Comuns (CTE)

Descrição

Pratique o uso de CTEs para simplificar consultas complexas.

Requisitos

- CTE básica
- CTE com múltiplas referências
- CTE recursiva
- CTE com operações DML

Solução

```
-- CTE básica
WITH pedidos_recentes AS (
    SELECT *
    FROM pratica.pedidos
    WHERE data_pedido >= CURRENT_DATE - INTERVAL '30 days'
)
SELECT
    c.nome as cliente,
    COUNT(pr.id) as total_pedidos_recentes
FROM pratica.clientes c
LEFT JOIN pedidos_recentes pr ON c.id = pr.cliente_id
GROUP BY c.nome;

-- CTE com múltiplas referências
WITH
    pedidos_por_cliente AS (
        SELECT
            cliente_id,
            COUNT(*) as total_pedidos,
            SUM(valor_total) as valor_total
        FROM pratica.pedidos
        GROUP BY cliente_id
    ),
```

```

    clientes_vip AS (
        SELECT cliente_id
        FROM pedidos_por_cliente
        WHERE total_pedidos >= 5 OR valor_total >= 5000
    )
SELECT
    c.nome,
    c.email,
    'VIP' as status
FROM pratica.clientes c
JOIN clientes_vip vip ON c.id = vip.cliente_id;

-- CTE recursiva
WITH RECURSIVE hierarquia_categorias AS (
    -- Caso base: categorias raiz (sem pai)
    SELECT id, nome, 0 as nivel
    FROM pratica.categorias
    WHERE categoria_pai_id IS NULL

    UNION ALL

    -- Caso recursivo: categorias filhas
    SELECT c.id, c.nome, h.nivel + 1
    FROM pratica.categorias c
    JOIN hierarquia_categorias h ON c.categoria_pai_id = h.id
)
SELECT
    REPEAT(' ', nivel) || nome as categoria_hierarquia,
    nivel
FROM hierarquia_categorias
ORDER BY nivel, nome;

-- CTE com operações DML
WITH produtos_sem_movimento AS (
    SELECT p.id
    FROM pratica.produtos p
    LEFT JOIN pratica.itens_pedido ip ON p.id = ip.produto_id
    WHERE ip.produto_id IS NULL
)

```

```

        AND p.data_cadastro < CURRENT_DATE - INTERVAL '180 days'
    )
UPDATE pratica.produtos
SET estoque = 0
WHERE id IN (SELECT id FROM produtos_sem_movimento);

```

Exercício 5: Operações com MERGE (UPSERT)

Descrição

Pratique operações de inserção/atualização condicional.

Requisitos

- INSERT ... ON CONFLICT (PostgreSQL)
- MERGE (SQL Server/Oracle)
- Atualização condicional
- Inserção condicional

Solução

```

-- INSERT ... ON CONFLICT (PostgreSQL)
INSERT INTO pratica.produtos (codigo, nome, preco, estoque)
VALUES
    ('PROD001', 'Smartphone X', 1299.99, 50),
    ('PROD008', 'Câmera Digital', 899.99, 25)
ON CONFLICT (codigo)
DO UPDATE SET
    nome = EXCLUDED.nome,
    preco = EXCLUDED.preco,
    estoque = pratica.produtos.estoque + EXCLUDED.estoque;

-- Simulação de MERGE para bancos que não suportam nativamente
-- Atualização condicional
WITH dados_novos (codigo, nome, preco, estoque) AS (

```

```

VALUES
    ( 'PROD001', 'Smartphone X Pro', 1499.99, 10),
    ( 'PROD009', 'Smartwatch', 399.99, 30)
)
UPDATE pratica.produtos p
SET
    nome = d.nome,
    preco = d.preco,
    estoque = p.estoque + d.estoque
FROM dados_novos d
WHERE p.codigo = d.codigo;

-- Inserção condicional
WITH dados_novos (codigo, nome, preco, estoque) AS (
VALUES
    ( 'PROD001', 'Smartphone X Pro', 1499.99, 10),
    ( 'PROD009', 'Smartwatch', 399.99, 30)
),
atualizados AS (
    UPDATE pratica.produtos p
    SET
        nome = d.nome,
        preco = d.preco,
        estoque = p.estoque + d.estoque
    FROM dados_novos d
    WHERE p.codigo = d.codigo
    RETURNING p.codigo
)
INSERT INTO pratica.produtos (codigo, nome, preco, estoque)
SELECT d.codigo, d.nome, d.preco, d.estoque
FROM dados_novos d
WHERE NOT EXISTS (
    SELECT 1 FROM atualizados a
    WHERE a.codigo = d.codigo
);

```

Exercício 6: Manipulação de Dados Temporais

Descrição

Pratique operações DML com dados temporais.

Requisitos

- Filtros por data/hora
- Cálculos com intervalos de tempo
- Agrupamento por períodos
- Comparações temporais

Solução

```
-- Filtros por data/hora
SELECT *
FROM pratica.pedidos
WHERE
    data_pedido >= CURRENT_DATE - INTERVAL '30 days'
    AND EXTRACT(HOUR FROM data_pedido) BETWEEN 9 AND 18;

-- Cálculos com intervalos de tempo
SELECT
    id,
    data_pedido,
    data_entrega,
    data_entrega - data_pedido AS tempo_processamento,
    EXTRACT(DAY FROM (data_entrega - data_pedido)) AS
dias_processamento
FROM pratica.pedidos
WHERE status = 'Entregue';

-- Agrupamento por períodos
SELECT
    DATE_TRUNC('month', data_pedido) AS mes,
```

```

    COUNT(*) AS total_pedidos,
    SUM(valor_total) AS valor_total,
    AVG(valor_total) AS ticket_medio
FROM pratica.pedidos
WHERE data_pedido >= CURRENT_DATE - INTERVAL '1 year'
GROUP BY mes
ORDER BY mes;

-- Comparações temporais
UPDATE pratica.pedidos
SET status = 'Atrasado'
WHERE
    status = 'Em Processamento'
    AND data_pedido < CURRENT_DATE - INTERVAL '3 days';

```

Exercício 7: Operações com JSON

Descrição

Pratique operações DML com dados em formato JSON.

Requisitos

- Consulta de dados JSON
- Atualização de dados JSON
- Filtragem por valores JSON
- Transformação JSON para tabela

Solução

```

-- Consulta de dados JSON
SELECT
    id,
    nome,
    dados_adicionais->>'endereco' AS endereco,

```

```

dados_adicionais->'contatos'->>'telefone' AS telefone
FROM pratica.clientes
WHERE dados_adicionais->'tipo' = 'Pessoa Física';

-- Atualização de dados JSON
UPDATE pratica.clientes
SET dados_adicionais = jsonb_set(
    dados_adicionais,
    '{endereco}',
    '"Rua Nova, 123"'::jsonb
)
WHERE id = 1;

-- Filtragem por valores JSON
SELECT *
FROM pratica.produtos
WHERE
    dados_tecnicos->'dimensoes'->>'altura' > '10'
    AND dados_tecnicos->'compatibilidade' ? 'Android';

-- Transformação JSON para tabela
SELECT
    p.id,
    p.nome,
    a.valor->>'nome' AS atributo_nome,
    a.valor->>'valor' AS atributo_valor
FROM
    pratica.produtos p,
    jsonb_array_elements(p.atributos) WITH ORDINALITY AS a(valor,
idx)
ORDER BY p.id, a.idx;

```

Exercício 8: Operações com Dados Geoespaciais

Descrição

Pratique operações DML com dados geoespaciais.

Requisitos

- Consulta por proximidade
- Cálculo de distâncias
- Filtragem por região
- Agrupamento por área

Solução

```
-- Consulta por proximidade
SELECT
    id,
    nome,
    ST_Distance(
        localizacao,
        ST_SetSRID(ST_MakePoint(-23.5505, -46.6333), 4326)
    ) AS distancia_km
FROM pratica.lojas
ORDER BY distancia_km
LIMIT 5;

-- Cálculo de distâncias
UPDATE pratica.entregas
SET
    distancia_km = ST_Distance(
        origem::geography,
        destino::geography
    ) / 1000,
    valor_frete = (ST_Distance(
        origem::geography,
        destino::geography
    ) / 1000) * 2.50 -- R$ 2,50 por km
WHERE status = 'Pendente';

-- Filtragem por região
```

```

SELECT
    c.nome,
    c.email
FROM practica.clientes c
WHERE ST_Contains(
    (SELECT area FROM practica.regioes WHERE nome = 'Zona Sul'),
    c.endereco_geo
);

-- Agrupamento por área
SELECT
    r.nome AS regiao,
    COUNT(c.id) AS total_clientes,
    AVG(c.limite_credito) AS limite_medio
FROM practica.regioes r
JOIN practica.clientes c ON ST_Contains(r.area, c.endereco_geo)
GROUP BY r.nome
ORDER BY total_clientes DESC;

```

Critérios de Avaliação

- Sintaxe correta e otimizada
- Uso adequado de recursos intermediários
- Eficiência das consultas
- Integridade dos dados após manipulação
- Aplicação de boas práticas

Dicas de Estudo

- Analise o plano de execução das consultas
- Compare diferentes abordagens para o mesmo problema

- Teste com conjuntos de dados maiores
- Verifique o impacto de índices nas consultas
- Pratique a escrita de consultas sem consultar exemplos

Exercícios Avançados de DML

Exercício 1: Operações de Sincronização de Dados

Descrição

Pratique operações de sincronização entre tabelas e esquemas.

Requisitos

- Sincronização bidirecional
- Detecção e resolução de conflitos
- Sincronização seletiva
- Registro de auditoria

Solução

```
-- Sincronização bidirecional
WITH
    origem_alteracoes AS (
        SELECT id, nome, preco, estoque, ultima_atualizacao
        FROM pratica.produtos_origem
        WHERE ultima_atualizacao > (SELECT
            MAX(ultima_sincronizacao) FROM pratica.controle_sync)
    ),
    destino_alteracoes AS (
        SELECT id, nome, preco, estoque, ultima_atualizacao
        FROM pratica.produtos_destino
        WHERE ultima_atualizacao > (SELECT
            MAX(ultima_sincronizacao) FROM pratica.controle_sync)
    ),
    conflitos AS (
        SELECT o.id
        FROM origem_alteracoes o
        JOIN destino_alteracoes d ON o.id = d.id
        WHERE o.ultima_atualizacao > d.ultima_atualizacao
    )
    -- Implementar lógica para inserir, atualizar ou deletar registros nos conflitos
```

```

),
atualizacoes_origem AS (
    UPDATE pratica.produtos_destino pd
    SET
        nome = po.nome,
        preco = po.preco,
        estoque = po.estoque,
        ultima_atualizacao = CURRENT_TIMESTAMP
    FROM pratica.produtos_origem po
    WHERE pd.id = po.id
    AND po.ultima_atualizacao > pd.ultima_atualizacao
    AND po.id NOT IN (SELECT id FROM conflitos)
    RETURNING pd.id
),
atualizacoes_destino AS (
    UPDATE pratica.produtos_origem po
    SET
        nome = pd.nome,
        preco = pd.preco,
        estoque = pd.estoque,
        ultima_atualizacao = CURRENT_TIMESTAMP
    FROM pratica.produtos_destino pd
    WHERE po.id = pd
    AND pd.ultima_atualizacao > po.ultima_atualizacao
    AND pd.id NOT IN (SELECT id FROM conflitos)
    RETURNING po.id
),
insercoes_origem AS (
    INSERT INTO pratica.produtos_destino (id, nome, preco,
estoque, ultima_atualizacao)
    SELECT id, nome, preco, estoque, ultima_atualizacao
    FROM origem.Alteracoes
    WHERE id NOT IN (SELECT id FROM destino.Alteracoes)
    RETURNING id
),
insercoes_destino AS (
    INSERT INTO pratica.produtos_origem (id, nome, preco,
estoque, ultima_atualizacao)

```

```

        SELECT id, nome, preco, estoque, ultima_atualizacao
        FROM destino.Alteracoes
        WHERE id NOT IN (SELECT id FROM origem.Alteracoes)
        RETURNING id
    )
SELECT
    'Atualizou produtos origem' AS operacao,
    atualizacoes_origem
FROM atualizacoes_origem
UNION ALL
SELECT
    'Atualizou produtos destino' AS operacao,
    atualizacoes_destino
FROM atualizacoes_destino
UNION ALL
SELECT
    'Inseriu produtos origem' AS operacao,
    insercoes_origem
FROM insercoes_origem
UNION ALL
SELECT
    'Inseriu produtos destino' AS operacao,
    insercoes_destino
FROM insercoes_destino;

-- Atualização do controle de sincronização
UPDATE pratica.controle_sync
SET ultima_sincronizacao = CURRENT_TIMESTAMP;

```

Exercício 2: Manipulação de Dados JSON/JSONB

Descrição

Trabalhe com armazenamento e manipulação de dados em formato JSON.

Requisitos

- Armazenamento de estruturas complexas
- Consultas em dados JSON
- Modificação de estruturas JSON
- Indexação de campos JSON

Solução

```
-- Tabela com dados JSON
CREATE TABLE pratica.configuracoes (
    id SERIAL PRIMARY KEY,
    aplicacao VARCHAR(50) NOT NULL,
    config JSONB NOT NULL,
    ultima_modificacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Inserção de dados JSON
INSERT INTO pratica.configuracoes (aplicacao, config)
VALUES (
    'sistema_vendas',
    '{
        "database": {
            "host": "db.example.com",
            "port": 5432,
            "credentials": {
                "username": "app_user",
                "password": "encrypted:abc123"
            }
        },
        "features": ["vendas", "estoque", "relatorios"],
        "limits": {
            "max_users": 100,
            "max_transactions": 10000
        },
        "logging": {
            "level": "info",
            "retention_days": 30
        }
    }'
);
```

```

        }
    }'::jsonb
);

-- Consultas em dados JSON
SELECT
    aplicacao,
    config->'database'->>'host' as db_host,
    config->'database'->>'port' as db_port,
    config->'features' as recursos_ativos,
    jsonb_array_length(config->'features') as total_recursos
FROM pratica.configuracoes
WHERE config->'limits'->>'max_users' = '100';

-- Atualização de dados JSON
UPDATE pratica.configuracoes
SET
    config = jsonb_set(
        jsonb_set(
            config,
            '{database,host}',
            '"new-db.example.com"'
        ),
        '{limits,max_users}',
        '200'
    ),
    ultima_modificacao = CURRENT_TIMESTAMP
WHERE aplicacao = 'sistema_vendas';

-- Adição de elementos em arrays JSON
UPDATE pratica.configuracoes
SET
    config = jsonb_set(
        config,
        '{features}',
        config->'features' || '"financeiro"'::jsonb
    ),
    ultima_modificacao = CURRENT_TIMESTAMP

```

```
WHERE aplicacao = 'sistema_vendas';

-- Índice para consultas em JSON
CREATE INDEX idx_config_features ON pratica.configuracoes
USING gin (config->'features');
```

Exercício 3: Operações com Window Functions

Descrição

Utilize window functions para análises avançadas de dados.

Requisitos

- Cálculo de métricas por janela
- Ranking e particionamento
- Análises cumulativas
- Comparações com períodos anteriores

Solução

```
-- Tabela de vendas para análise
CREATE TABLE IF NOT EXISTS pratica.vendas_mensais (
    id SERIAL PRIMARY KEY,
    produto_id INTEGER NOT NULL,
    categoria_id INTEGER NOT NULL,
    regiao VARCHAR(50) NOT NULL,
    data_venda DATE NOT NULL,
    quantidade INTEGER NOT NULL,
    valor_total DECIMAL(12,2) NOT NULL
);

-- Ranking de produtos por região
SELECT
    regiao,
```

```

produto_id,
SUM(valor_total) as total_vendas,
RANK() OVER (PARTITION BY regiao ORDER BY SUM(valor_total)
DESC) as rankingRegional,
DENSE_RANK() OVER (ORDER BY SUM(valor_total) DESC) as
ranking_geral
FROM practica.vendas_mensais
GROUP BY regiao, produto_id;

-- Análise de tendências com janelas móveis
SELECT
data_venda,
categoria_id,
SUM(valor_total) as vendas_diaris,
AVG(SUM(valor_total)) OVER (
    PARTITION BY categoria_id
    ORDER BY data_venda
    ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
) as media_movel_7dias,
SUM(SUM(valor_total)) OVER (
    PARTITION BY categoria_id
    ORDER BY data_venda
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
) as vendas_acumuladas
FROM practica.vendas_mensais
GROUP BY data_venda, categoria_id
ORDER BY categoria_id, data_venda;

-- Comparação com período anterior
SELECT
EXTRACT(YEAR FROM data_venda) as ano,
EXTRACT(MONTH FROM data_venda) as mes,
SUM(valor_total) as vendas_mes,
LAG(SUM(valor_total), 1) OVER (
    PARTITION BY EXTRACT(MONTH FROM data_venda)
    ORDER BY EXTRACT(YEAR FROM data_venda)
) as vendas_mes_ano_anterior,
SUM(valor_total) - LAG(SUM(valor_total), 1) OVER (

```

```

        PARTITION BY EXTRACT(MONTH FROM data_venda)
        ORDER BY EXTRACT(YEAR FROM data_venda)
    ) as diferenca,
CASE
    WHEN LAG(SUM(valor_total), 1) OVER (
        PARTITION BY EXTRACT(MONTH FROM data_venda)
        ORDER BY EXTRACT(YEAR FROM data_venda)
    ) > 0 THEN
        ROUND(
            (SUM(valor_total) - LAG(SUM(valor_total), 1) OVER
(
        PARTITION BY EXTRACT(MONTH FROM data_venda)
        ORDER BY EXTRACT(YEAR FROM data_venda)
    )) / LAG(SUM(valor_total), 1) OVER (
        PARTITION BY EXTRACT(MONTH FROM data_venda)
        ORDER BY EXTRACT(YEAR FROM data_venda)
    ) * 100
        )
    ELSE NULL
END as variacao_percentual
FROM pratica.vendas_mensais
GROUP BY ano, mes
ORDER BY ano, mes;

```

Exercício 4: Operações com Common Table Expressions (CTEs)

Descrição

Utilize CTEs para consultas complexas e recursivas.

Requisitos

- CTEs para modularização de consultas
- CTEs recursivas para estruturas hierárquicas

- CTEs para operações em múltiplas etapas
- Análise de caminhos em grafos

Solução

```
-- Tabela de categorias hierárquicas
CREATE TABLE IF NOT EXISTS pratica.categorias (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    categoria_pai_id INTEGER REFERENCES pratica.categorias(id)
);

-- CTE recursiva para hierarquia de categorias
WITH RECURSIVE hierarquia AS (
    -- Caso base: categorias raiz
    SELECT
        id,
        nome,
        categoria_pai_id,
        1 as nivel,
        ARRAY[id] as caminho,
        nome as caminho_nome
    FROM pratica.categorias
    WHERE categoria_pai_id IS NULL

    UNION ALL

    -- Caso recursivo: categorias filhas
    SELECT
        c.id,
        c.nome,
        c.categoria_pai_id,
        h.nivel + 1,
        h.caminho || c.id,
        h.caminho_nome || ' > ' || c.nome
    FROM pratica.categorias c
    JOIN hierarquia h ON c.categoria_pai_id = h.id
)
```

```

)
SELECT
    id,
    nome,
    nivel,
    caminho,
    caminho_nome
FROM hierarquia
ORDER BY caminho;

-- CTE para análise de vendas em múltiplas etapas
WITH vendas_mensais AS (
    SELECT
        EXTRACT(YEAR FROM data_venda) as ano,
        EXTRACT(MONTH FROM data_venda) as mes,
        categoria_id,
        SUM(valor_total) as total_vendas
    FROM pratica.vendas_mensais
    GROUP BY ano, mes, categoria_id
),
ranking_mensal AS (
    SELECT
        ano,
        mes,
        categoria_id,
        total_vendas,
        RANK() OVER (PARTITION BY ano, mes ORDER BY total_vendas
DESC) as ranking
    FROM vendas_mensais
),
top_categorias AS (
    SELECT
        ano,
        mes,
        categoria_id,
        total_vendas
    FROM ranking_mensal
    WHERE ranking <= 3
)

```

```

)
SELECT
    tc.ano,
    tc.mes,
    tc.categoria_id,
    c.nome as categoria,
    tc.total_vendas,
    ROUND(tc.total_vendas / SUM(vm.total_vendas) OVER (PARTITION
    BY tc.ano, tc.mes) * 100, 2) as percentual_total
FROM top_categorias tc
JOIN pratica.categorias c ON tc.categoria_id = c.id
JOIN vendas_mensais vm ON tc.ano = vm.ano AND tc.mes = vm.mes
ORDER BY tc.ano, tc.mes, tc.total_vendas DESC;

```

Exercício 5: Operações com Pivotamento de Dados

Descrição

Transforme dados de formato linha para coluna e vice-versa.

Requisitos

- Pivotamento dinâmico de linhas para colunas
- Despivotamento de colunas para linhas
- Agregações em pivotamento
- Formatação de relatórios

Solução

```

-- Tabela de vendas por região e produto
CREATE TABLE IF NOT EXISTS pratica.vendas_regiao (
    id SERIAL PRIMARY KEY,
    data_venda DATE NOT NULL,
    regiao VARCHAR(50) NOT NULL,
    produto_id INTEGER NOT NULL,

```

```

        quantidade INTEGER NOT NULL,
        valor_total DECIMAL(12,2) NOT NULL
);

-- Pivotamento de linhas para colunas (vendas por região)
SELECT
    data_venda,
    produto_id,
    SUM(CASE WHEN regiao = 'Norte' THEN valor_total ELSE 0 END) as norte,
    SUM(CASE WHEN regiao = 'Sul' THEN valor_total ELSE 0 END) as sul,
    SUM(CASE WHEN regiao = 'Leste' THEN valor_total ELSE 0 END) as leste,
    SUM(CASE WHEN regiao = 'Oeste' THEN valor_total ELSE 0 END) as oeste,
    SUM(valor_total) as total_geral
FROM practica.vendas_regiao
GROUP BY data_venda, produto_id
ORDER BY data_venda, produto_id;

-- Pivotamento dinâmico usando crosstab (PostgreSQL)
CREATE EXTENSION IF NOT EXISTS tablefunc;

SELECT * FROM crosstab(
    'SELECT
        produto_id,
        regiao,
        SUM(valor_total)
    FROM practica.vendas_regiao
    WHERE data_venda BETWEEN ''2023-01-01'' AND ''2023-01-31''
    GROUP BY produto_id, regiao
    ORDER BY 1, 2',
    'SELECT DISTINCT regiao FROM practica.vendas_regiao ORDER BY 1'
) AS ct (
    produto_id INTEGER,
    norte DECIMAL,
    sul DECIMAL,

```

```

        leste DECIMAL,
        oeste DECIMAL
);

-- Despivotamento de colunas para linhas
CREATE TABLE IF NOT EXISTS pratica.relatorio_mensal (
    produto_id INTEGER NOT NULL,
    mes DATE NOT NULL,
    norte DECIMAL(12,2),
    sul DECIMAL(12,2),
    leste DECIMAL(12,2),
    oeste DECIMAL(12,2),
    PRIMARY KEY (produto_id, mes)
);

-- Despivotamento (PostgreSQL)
SELECT
    produto_id,
    mes,
    regiao,
    valor
FROM pratica.relatorio_mensal
CROSS JOIN LATERAL (
    VALUES
        ('Norte', norte),
        ('Sul', sul),
        ('Leste', leste),
        ('Oeste', oeste)
) as regioes(regiao, valor)
WHERE valor > 0
ORDER BY produto_id, mes, regiao;

```

Exercício 6: Operações com Dados Temporais

Descrição

Trabalhe com análises temporais avançadas.

Requisitos

- Análise de séries temporais
- Cálculo de períodos de negócio
- Detecção de padrões temporais
- Agregações por intervalos customizados

Solução

```
-- Tabela de eventos temporais
CREATE TABLE IF NOT EXISTS pratica.eventos_sistema (
    id SERIAL PRIMARY KEY,
    tipo_evento VARCHAR(50) NOT NULL,
    timestamp TIMESTAMP NOT NULL,
    usuario_id INTEGER,
    duracao_ms INTEGER,
    detalhes JSONB
);

-- Análise de distribuição por hora do dia
SELECT
    EXTRACT(HOUR FROM timestamp) as hora,
    COUNT(*) as total_eventos,
    AVG(duracao_ms) as duracao_media,
    COUNT(*) * 100.0 / SUM(COUNT(*)) OVER () as percentual
FROM pratica.eventos_sistema
WHERE timestamp >= CURRENT_DATE - INTERVAL '30 days'
GROUP BY hora
ORDER BY hora;

-- Cálculo de tempo entre eventos (lag/lead)
SELECT
    id,
    tipo_evento,
    timestamp,
```

```

usuario_id,
EXTRACT(EPOCH FROM (timestamp - LAG(timestamp) OVER (
    PARTITION BY usuario_id
    ORDER BY timestamp
))) as segundos_desde_ultimo_evento,
EXTRACT(EPOCH FROM (LEAD(timestamp) OVER (
    PARTITION BY usuario_id
    ORDER BY timestamp
) - timestamp)) as segundos_ate_proximo_evento
FROM practica.eventos_sistema
WHERE usuario_id = 123
ORDER BY timestamp;

-- Agregação por intervalos customizados (15 minutos)
SELECT
    DATE_TRUNC('day', timestamp) as dia,
    (EXTRACT(HOUR FROM timestamp) * 4 +
     FLOOR(EXTRACT(MINUTE FROM timestamp) / 15)) * 15 as
minuto_do_dia,
    COUNT(*) as total_eventos,
    SUM(duracao_ms) as duracao_total
FROM practica.eventos_sistema
WHERE timestamp >= CURRENT_DATE - INTERVAL '7 days'
GROUP BY dia, minuto_do_dia
ORDER BY dia, minuto_do_dia;

-- Detecção de padrões temporais (3 falhas em 5 minutos)
WITH eventos_falha AS (
    SELECT
        timestamp,
        usuario_id,
        tipo_evento,
        COUNT(*) OVER (
            PARTITION BY usuario_id
            ORDER BY timestamp
            RANGE BETWEEN INTERVAL '5 minutes' PRECEDING AND
CURRENT ROW
        ) as falhas_recentes
)

```

```

        FROM pratica.eventos_sistema
        WHERE tipo_evento = 'login_falha'
    )
SELECT DISTINCT
    usuario_id,
    MIN(timestamp) OVER (PARTITION BY usuario_id) as
primeira_falha,
    COUNT(*) OVER (PARTITION BY usuario_id) as total_falhas
FROM eventos_falha
WHERE falhas_recentes >= 3
ORDER BY usuario_id;

```

Exercício 7: Operações com Dados Geoespaciais

Descrição

Trabalhe com dados geoespaciais para análises baseadas em localização.

Requisitos

- Armazenamento de coordenadas
- Cálculo de distâncias
- Consultas por proximidade
- Análises de áreas

Solução

```

-- Extensão PostGIS (PostgreSQL)
CREATE EXTENSION IF NOT EXISTS postgis;

-- Tabela de locais
CREATE TABLE IF NOT EXISTS pratica.locais (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    tipo VARCHAR(50) NOT NULL,

```

```

geom GEOMETRY(Point, 4326) NOT NULL
);

-- Inserção de locais
INSERT INTO pratica.locais (nome, tipo, geom)
VALUES
    ('Parque Central', 'Parque', ST_SetSRID(ST_MakePoint(-46.6333,
-23.5505), 4326)),
    ('Museu de Arte', 'Museu', ST_SetSRID(ST_MakePoint(-46.6358,
-23.5515), 4326)),
    ('Teatro Municipal', 'Teatro',
ST_SetSRID(ST_MakePoint(-46.6345, -23.5520), 4326));

-- Consulta de locais próximos
SELECT
    nome,
    tipo,
    ST_Distance(geom, ST_SetSRID(ST_MakePoint(-46.6333, -23.5505),
4326)) as distancia_km
FROM pratica.locais
ORDER BY distancia_km;

-- Cálculo de área de um polígono
CREATE TABLE IF NOT EXISTS pratica.poligonos (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    geom GEOMETRY(Polygon, 4326) NOT NULL
);

INSERT INTO pratica.poligonos (nome, geom)
VALUES (
    'Área Turística',
    ST_SetSRID(ST_GeomFromText('POLYGON((-46.6333 -23.5505,
-46.6358 -23.5515, -46.6345 -23.5520, -46.6333 -23.5505))'), 4326)
);

SELECT
    nome,

```

```
ST_Area(geom) as area_km2  
FROM practica.poligonos;
```

Laboratórios de DML

Visão Geral

Os laboratórios práticos de DML (Data Manipulation Language) são projetados para fornecer experiência hands-on com operações de manipulação de dados em ambientes PostgreSQL. Cada laboratório apresenta cenários reais e desafios práticos comumente encontrados em ambientes de produção.

Estrutura dos Laboratórios

Cada laboratório segue uma estrutura consistente:

1. Preparação do Ambiente

- Configuração inicial
- Pré-requisitos
- Scripts de setup

2. Objetivos de Aprendizado

- Conceitos principais
- Habilidades técnicas
- Resultados esperados

3. Roteiro Prático

- Instruções passo a passo
- Comandos e scripts
- Pontos de verificação

4. Avaliação

- Critérios de conclusão

- Testes de validação
- Métricas de sucesso

Ambiente de Laboratório

```
-- Criar database dedicado para laboratórios
CREATE DATABASE lab_dml;

-- Schema para isolamento de exercícios
CREATE SCHEMA lab_workspace;

-- Tabelas para os laboratórios
CREATE TABLE lab_workspace.clientes (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    data_cadastro DATE DEFAULT CURRENT_DATE,
    status VARCHAR(20) DEFAULT 'Ativo',
    segmento VARCHAR(50),
    limite_credito DECIMAL(10, 2) DEFAULT 1000.00
);

CREATE TABLE lab_workspace.produtos (
    id SERIAL PRIMARY KEY,
    codigo VARCHAR(20) UNIQUE,
    nome VARCHAR(100) NOT NULL,
    descricao TEXT,
    preco DECIMAL(10, 2) NOT NULL CHECK (preco > 0),
    estoque INTEGER DEFAULT 0,
    categoria VARCHAR(50),
    data_cadastro DATE DEFAULT CURRENT_DATE,
    status VARCHAR(20) DEFAULT 'Ativo'
);

CREATE TABLE lab_workspace_pedidos (
    id SERIAL PRIMARY KEY,
```

```

cliente_id INTEGER REFERENCES lab_workspace.clientes(id),
data_pedido TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
valor_total DECIMAL(10,2) DEFAULT 0,
status VARCHAR(20) DEFAULT 'Pendente',
observacoes TEXT
);

CREATE TABLE lab_workspace.itens_pedido (
    pedido_id INTEGER REFERENCES lab_workspace.pedidos(id),
    produto_id INTEGER REFERENCES lab_workspace.produtos(id),
    quantidade INTEGER CHECK (quantidade > 0),
    preco_unitario DECIMAL(10,2) NOT NULL,
    desconto DECIMAL(10,2) DEFAULT 0,
    PRIMARY KEY (pedido_id, produto_id)
);

-- Tabela de controle de progresso
CREATE TABLE lab_workspace.lab_progress (
    lab_id SERIAL PRIMARY KEY,
    lab_name VARCHAR(100),
    start_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    completion_time TIMESTAMP,
    status VARCHAR(20) DEFAULT 'IN_PROGRESS',
    notes TEXT
);

```

Laboratórios Disponíveis

1. Manipulação de Dados

- Operações CRUD básicas
- Transações e consistência
- Manipulação em lote
- Validação de dados

2. Análise de Dados

- Consultas analíticas
- Agregações e agrupamentos
- Funções de janela
- Relatórios dinâmicos

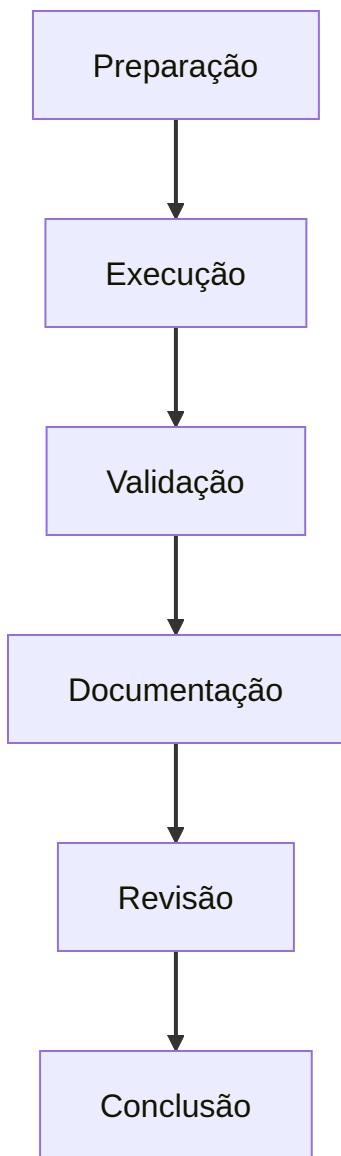
3. Transformação de Dados

- Conversão de formatos
- Limpeza de dados
- Normalização e desnormalização
- Enriquecimento de dados

4. Integração de Dados

- Importação e exportação
- Sincronização entre tabelas
- Migração de dados
- Validação de integridade

Fluxo de Trabalho Recomendado



Boas Práticas

- | | | |
|--|--|--|
| | DIRETRIZES DE LABORATÓRIO: | |
| | ✓ Sempre trabalhar em transações | |
| | ✓ Verificar resultados após cada etapa | |
| | ✓ Documentar todas as operações | |
| | ✓ Testar com diferentes conjuntos | |
| | ✓ Manter scripts de rollback | |

Ferramentas Necessárias

1. PostgreSQL Client

- psql
- pgAdmin 4
- DBeaver

2. Utilitários

- pg_dump (para backup)
- COPY (para importação/exportação)
- Ferramentas de análise de plano de execução

3. Scripts de Suporte

```
-- Script de verificação de ambiente
CREATE OR REPLACE FUNCTION lab_workspace.check_environment()
RETURNS TABLE (
    check_item VARCHAR,
    status VARCHAR,
    details TEXT
) AS $$
BEGIN
    RETURN QUERY
        SELECT 'Database Version'::VARCHAR,
               version()::VARCHAR,
               'Verificação da versão do PostgreSQL' ::TEXT
        UNION ALL
        SELECT 'Available Tables',
               (SELECT string_agg(table_name, ', ') FROM
information_schema.tables
               WHERE table_schema = 'lab_workspace'),
               'Tabelas disponíveis no schema lab_workspace';

```

```
END;  
$$ LANGUAGE plpgsql;
```

Navegação dos Laboratórios

- Laboratório de Manipulação de Dados ([Laboratório: Manipulação de Dados](#))
- Laboratório de Análise de Dados ([Laboratório: Análise de Dados](#))
- Laboratório de Transformação de Dados ([Laboratório: Transformação de Dados](#))
- Laboratório de Integração de Dados ([Laboratório: Integração de Dados](#))

Supporte e Recursos

Documentação

- PostgreSQL Official Documentation (<https://www.postgresql.org/docs/>)
- DML Reference Guide (<https://www.postgresql.org/docs/current/dml.html>)
- Data Manipulation Best Practices (<https://www.postgresql.org/docs/current/sql-commands.html>)

Comunidade

- Fórum PostgreSQL
- Stack Overflow
- GitHub Discussions

Conclusão

Os laboratórios DML fornecem uma base prática essencial para o desenvolvimento de habilidades em manipulação de dados. A prática regular destes exercícios contribuirá significativamente para sua expertise em operações de banco de dados PostgreSQL.

Próximos Passos

1. Preparação

- Configure seu ambiente local
- Revise os pré-requisitos
- Familiarize-se com as ferramentas

2. Execução

- Siga os laboratórios em ordem
- Complete todos os exercícios
- Documente seus resultados

3. Avançado

- Explore variações dos exercícios
- Crie seus próprios cenários
- Compartilhe experiências



Nota: Certifique-se de manter backups e usar ambientes de teste apropriados durante a execução dos laboratórios. As operações DML podem modificar dados permanentemente.

Laboratório: Manipulação de Dados

Objetivo

Praticar operações DML (Data Manipulation Language) básicas em um banco de dados relacional, aplicando boas práticas de manipulação de dados.

Cenário

Você é um desenvolvedor de banco de dados em uma empresa de e-commerce que precisa implementar operações de manipulação de dados para o sistema de gerenciamento de pedidos.

Setup Inicial

```
-- Carregar dados de exemplo
INSERT INTO lab_workspace.clientes (nome, email, segmento,
limite_credito)
VALUES
    ('Ana Silva', 'ana.silva@email.com', 'Varejo', 2000.00),
    ('Bruno Costa', 'bruno@email.com', 'Atacado', 5000.00),
    ('Carla Mendes', 'carla@email.com', 'Varejo', 1500.00),
    ('Daniel Oliveira', 'daniel@email.com', 'Corporativo',
10000.00),
    ('Elena Santos', 'elena@email.com', 'Varejo', 2500.00);

INSERT INTO lab_workspace.produtos (codigo, nome, descricao,
preco, estoque, categoria)
VALUES
    ('PROD001', 'Smartphone X', 'Smartphone avançado com câmera de
alta resolução', 1299.99, 50, 'Eletrônicos'),
    ('PROD002', 'Notebook Pro', 'Notebook para uso profissional',
3499.99, 20, 'Informática'),
    ('PROD003', 'Fones Bluetooth', 'Fones de ouvido sem fio',
```

```

199.99, 100, 'Acessórios'),
('PROD004', 'Monitor 24"', 'Monitor LED Full HD', 799.99, 30,
'Informática'),
('PROD005', 'Teclado Mecânico', 'Teclado para gamers', 249.99,
45, 'Periféricos');

-- Registrar início do laboratório
INSERT INTO lab_workspace.lab_progress (lab_name)
VALUES ('Manipulação de Dados');

```

Parte 1: Operações CRUD Básicas

1.1 Inserção de Dados

```

-- Inserir um novo cliente
INSERT INTO lab_workspace.clientes (nome, email, segmento,
limite_credito)
VALUES ('Fernando Gomes', 'fernando@email.com', 'Varejo',
3000.00);

-- Inserir múltiplos produtos
INSERT INTO lab_workspace.produtos (codigo, nome, descricao,
preco, estoque, categoria)
VALUES
    ('PROD006', 'Mouse Sem Fio', 'Mouse ergonômico', 89.90, 60,
'Periféricos'),
    ('PROD007', 'Caixa de Som', 'Caixa de som bluetooth', 159.90,
25, 'Acessórios');

-- Inserir um pedido com seus itens
INSERT INTO lab_workspace_pedidos (cliente_id, observacoes)
VALUES (1, 'Entrega prioritária');

INSERT INTO lab_workspace_itens_pedido (pedido_id, produto_id,
quantidade, preco_unitario)
VALUES
    (1, 1, 1, 1299.99),

```

```

(1, 3, 2, 199.99);

UPDATE lab_workspace.pedidos
SET valor_total = (
    SELECT SUM(quantidade * preco_unitario)
    FROM lab_workspace.itens_pedido
    WHERE pedido_id = 1
)
WHERE id = 1;

```

1.2 Consulta de Dados

```

-- Consulta simples
SELECT * FROM lab_workspace.clientes WHERE segmento = 'Varejo';

-- Consulta com JOIN
SELECT
    p.id as pedido_id,
    c.nome as cliente,
    p.data_pedido,
    p.valor_total,
    p.status
FROM lab_workspace.pedidos p
JOIN lab_workspace.clientes c ON p.cliente_id = c.id;

-- Consulta detalhada de pedido
SELECT
    p.id as pedido_id,
    c.nome as cliente,
    pr.nome as produto,
    ip.quantidade,
    ip.preco_unitario,
    (ip.quantidade * ip.preco_unitario) as subtotal
FROM lab_workspace.pedidos p
JOIN lab_workspace.clientes c ON p.cliente_id = c.id
JOIN lab_workspace.itens_pedido ip ON p.id = ip.pedido_id

```

```
JOIN lab_workspace.produtos pr ON ip.producto_id = pr.id  
WHERE p.id = 1;
```

1.3 Atualização de Dados

```
-- Atualização simples  
UPDATE lab_workspace.clientes  
SET email = 'ana.silva.nova@email.com'  
WHERE id = 1;  
  
-- Atualização com cálculo  
UPDATE lab_workspace.produtos  
SET preco = preco * 1.05  
WHERE categoria = 'Eletrônicos';  
  
-- Atualização com subconsulta  
UPDATE lab_workspace_pedidos  
SET status = 'Aprovado'  
WHERE cliente_id IN (  
    SELECT id FROM lab_workspace.clientes  
    WHERE limite_credito > 2000.00  
);
```

1.4 Exclusão de Dados

```
-- Exclusão simples  
DELETE FROM lab_workspace_itens_pedido  
WHERE pedido_id = 1 AND produto_id = 3;  
  
-- Exclusão com subconsulta  
DELETE FROM lab_workspace.produtos  
WHERE estoque = 0 AND id NOT IN (  
    SELECT produto_id FROM lab_workspace_itens_pedido  
);  
  
-- Exclusão em cascata
```

```
DELETE FROM lab_workspace.itens_pedido
WHERE pedido_id = 1;

DELETE FROM lab_workspace.pedidos
WHERE id = 1;
```

Parte 2: Manipulação em Lote

2.1 Inserção em Lote

```
-- Inserir múltiplos clientes
INSERT INTO lab_workspace.clientes (nome, email, segmento,
limite_credito)
SELECT
    'Cliente ' || i,
    'cliente' || i || '@email.com',
    CASE WHEN i % 3 = 0 THEN 'Varejo'
        WHEN i % 3 = 1 THEN 'Atacado'
        ELSE 'Corporativo' END,
    1000 * (i % 10 + 1)
FROM generate_series(1, 10) i;

-- Verificar inserção
SELECT COUNT(*) FROM lab_workspace.clientes;
```

2.2 Atualização em Lote

```
-- Atualizar status de múltiplos pedidos
UPDATE lab_workspace.pedidos
SET status = 'Processando'
WHERE status = 'Pendente';

-- Atualizar preços com base na categoria
UPDATE lab_workspace.produtos
```

```
SET preco = preco * 0.95 -- 5% desconto
WHERE categoria = 'Eletrônicos';
```

2.3 Exclusão em Lote

```
-- Excluir pedidos antigos
DELETE FROM lab_workspace.itens_pedido
WHERE pedido_id IN (
    SELECT id FROM lab_workspace.pedidos
    WHERE status = 'Cancelado'
);

DELETE FROM lab_workspace.pedidos
WHERE status = 'Cancelado';
```

Conclusão

Neste laboratório, você praticou as operações DML básicas:

- Inserção de dados (INSERT)
- Consulta de dados (SELECT)
- Atualização de dados (UPDATE)
- Exclusão de dados (DELETE)

Estas operações são fundamentais para qualquer sistema de banco de dados e formam a base para manipulação de dados em aplicações.

Laboratório: Análise de Dados

Objetivo

Desenvolver habilidades avançadas de análise de dados utilizando SQL, incluindo agregações, funções de janela, agrupamentos e técnicas de relatórios dinâmicos.

Cenário

Você é um analista de dados em uma empresa de varejo que precisa extrair insights valiosos dos dados de vendas, clientes e produtos para apoiar decisões estratégicas.

Setup Inicial

```
-- Criar tabelas adicionais para análise
CREATE TABLE lab_workspace.vendas (
    id SERIAL PRIMARY KEY,
    data_venda DATE NOT NULL,
    cliente_id INTEGER REFERENCES lab_workspace.clientes(id),
    vendedor_id INTEGER,
    loja_id INTEGER,
    valor_total DECIMAL(10,2) NOT NULL,
    desconto DECIMAL(10,2) DEFAULT 0,
    metodo_pagamento VARCHAR(50),
    parcelas INTEGER DEFAULT 1
);

CREATE TABLE lab_workspace.itens_venda (
    id SERIAL PRIMARY KEY,
    venda_id INTEGER REFERENCES lab_workspace.vendas(id),
    produto_id INTEGER REFERENCES lab_workspace.produtos(id),
    quantidade INTEGER NOT NULL,
    preco_unitario DECIMAL(10,2) NOT NULL
);

CREATE TABLE lab_workspace.lojas (
```

```

        id SERIAL PRIMARY KEY,
        nome VARCHAR(100) NOT NULL,
        regiao VARCHAR(50),
        tipo VARCHAR(50),
        data_inauguracao DATE,
        tamanho_m2 INTEGER
);

CREATE TABLE lab_workspace.vendedores (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    loja_id INTEGER REFERENCES lab_workspace.lojas(id),
    data_contratacao DATE,
    meta_mensal DECIMAL(10,2),
    status VARCHAR(20) DEFAULT 'Ativo'
);

-- Carregar dados de exemplo
INSERT INTO lab_workspace.lojas (nome, regiao, tipo,
data_inauguracao, tamanho_m2)
VALUES
    ('Loja Centro', 'Sul', 'Shopping', '2020-01-15', 500),
    ('Loja Norte', 'Norte', 'Rua', '2019-05-20', 350),
    ('Loja Shopping Plaza', 'Leste', 'Shopping', '2021-03-10',
600),
    ('Loja Beira Mar', 'Nordeste', 'Shopping', '2018-11-05', 450),
    ('Loja Parque', 'Centro-Oeste', 'Rua', '2022-02-28', 300);

INSERT INTO lab_workspace.vendedores (nome, email, loja_id,
data_contratacao, meta_mensal)
VALUES
    ('Carlos Souza', 'carlos@email.com', 1, '2020-02-01',
20000.00),
    ('Mariana Lima', 'mariana@email.com', 1, '2020-03-15',
18000.00),
    ('Paulo Mendes', 'paulo@email.com', 2, '2019-06-10',
15000.00),

```

```

        ('Juliana Costa', 'juliana@email.com', 3, '2021-04-05',
22000.00),
        ('Roberto Silva', 'roberto@email.com', 4, '2019-01-10',
19000.00),
        ('Amanda Oliveira', 'amanda@email.com', 5, '2022-03-01',
16000.00);

-- Inserir dados de vendas (últimos 6 meses)
INSERT INTO lab_workspace.vendas (data_venda, cliente_id,
vendedor_id, loja_id, valor_total, desconto, metodo_pagamento,
parcelas)
SELECT
    CURRENT_DATE - (random() * 180)::integer AS data_venda,
    (random() * 50 + 1)::integer AS cliente_id,
    (random() * 6 + 1)::integer AS vendedor_id,
    (random() * 5 + 1)::integer AS loja_id,
    (random() * 1000 + 100)::numeric(10,2) AS valor_total,
    (random() * 100)::numeric(10,2) AS desconto,
    (ARRAY['Crédito', 'Débito', 'Dinheiro', 'Pix', 'Boleto'])
[floor(random() * 5 + 1)] AS metodo_pagamento,
    (ARRAY[1, 2, 3, 6, 10, 12])[floor(random() * 6 + 1)] AS
parcelas
FROM generate_series(1, 1000);

-- Inserir itens de venda
INSERT INTO lab_workspace.itens_venda (venda_id, produto_id,
quantidade, preco_unitario)
SELECT
    v.id AS venda_id,
    (random() * 100 + 1)::integer AS produto_id,
    (random() * 5 + 1)::integer AS quantidade,
    (random() * 200 + 50)::numeric(10,2) AS preco_unitario
FROM lab_workspace.vendas v
CROSS JOIN generate_series(1, 3) AS num_items
ORDER BY v.id;

-- Registrar início do laboratório

```

```
INSERT INTO lab_workspace.lab_progress (lab_name)
VALUES ('Análise de Dados');
```

Parte 1: Agregações Básicas

1.1 Contagem e Somatórios

```
-- Total de vendas por loja
SELECT
    l.nome as loja,
    COUNT(v.id) as total_vendas,
    SUM(v.valor_total) as valor_total
FROM lab_workspace.vendas v
JOIN lab_workspace.lojas l ON v.loja_id = l.id
GROUP BY l.nome
ORDER BY valor_total DESC;

-- Média de vendas por dia da semana
SELECT
    TO_CHAR(data_venda, 'Day') as dia_semana,
    COUNT(id) as total_vendas,
    ROUND(AVG(valor_total), 2) as ticket_medio
FROM lab_workspace.vendas
GROUP BY dia_semana
ORDER BY ticket_medio DESC;

-- Produtos mais vendidos
SELECT
    p.nome as produto,
    SUM(iv.quantidade) as quantidade_total,
    SUM(iv.quantidade * iv.preco_unitario) as valor_total
FROM lab_workspace.itens_venda iv
JOIN lab_workspace.produtos p ON iv.producto_id = p.id
GROUP BY p.nome
```

```
ORDER BY quantidade_total DESC  
LIMIT 10;
```

1.2 Funções de Agregação Avançadas

```
-- Estatísticas de vendas  
  
SELECT  
    COUNT(*) as total_vendas,  
    MIN(valor_total) as menor_venda,  
    MAX(valor_total) as maior_venda,  
    ROUND(AVG(valor_total), 2) as media_vendas,  
    PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY valor_total) as  
mediana_vendas,  
    ROUND(STDDEV(valor_total), 2) as desvio_padrao  
FROM lab_workspace.vendas;  
  
-- Distribuição de vendas por faixa de valor  
  
SELECT  
    CASE  
        WHEN valor_total < 200 THEN 'Até R$ 200'  
        WHEN valor_total < 500 THEN 'R$ 200 a R$ 500'  
        WHEN valor_total < 800 THEN 'R$ 500 a R$ 800'  
        ELSE 'Acima de R$ 800'  
    END as faixa_valor,  
    COUNT(*) as total_vendas,  
    ROUND(AVG(valor_total), 2) as valor_medio,  
    SUM(valor_total) as valor_total  
FROM lab_workspace.vendas  
GROUP BY faixa_valor  
ORDER BY MIN(valor_total);  
  
-- Análise de métodos de pagamento  
  
SELECT  
    metodo_pagamento,  
    COUNT(*) as total_vendas,  
    ROUND(COUNT(*) * 100.0 / (SELECT COUNT(*) FROM  
lab_workspace.vendas), 2) as percentual,
```

```
    SUM(valor_total) as valor_total
FROM lab_workspace.vendas
GROUP BY metodo_pagamento
ORDER BY total_vendas DESC;
```

Parte 2: Análise Temporal

2.1 Tendências por Período

```
-- Vendas por mês
SELECT
    TO_CHAR(data_venda, 'YYYY-MM') as mes,
    COUNT(*) as total_vendas,
    SUM(valor_total) as valor_total,
    ROUND(AVG(valor_total), 2) as ticket_medio
FROM lab_workspace.vendas
GROUP BY mes
ORDER BY mes;

-- Vendas por semana
SELECT
    TO_CHAR(data_venda, 'YYYY-IW') as semana,
    COUNT(*) as total_vendas,
    SUM(valor_total) as valor_total
FROM lab_workspace.vendas
GROUP BY semana
ORDER BY semana;

-- Comparação dia a dia (últimos 14 dias)
SELECT
    data_venda,
    COUNT(*) as total_vendas,
    SUM(valor_total) as valor_total,
    LAG(SUM(valor_total), 1) OVER (ORDER BY data_venda) as
valor_dia_anterior,
    ROUND(
        (SUM(valor_total) - LAG(SUM(valor_total), 1) OVER (ORDER
```

```

    BY data_venda)) /
    NULLIF(LAG(SUM(valor_total), 1) OVER (ORDER BY
data_venda), 0) * 100,
    2) as variacao_percentual
FROM lab_workspace.vendas
WHERE data_venda >= CURRENT_DATE - INTERVAL '14 days'
GROUP BY data_venda
ORDER BY data_venda;

```

2.2 Análise de Sazonalidade

```

-- Vendas por hora do dia
SELECT
    EXTRACT(HOUR FROM data_venda::timestamp) as hora,
    COUNT(*) as total_vendas,
    SUM(valor_total) as valor_total
FROM lab_workspace.vendas
GROUP BY hora
ORDER BY hora;

-- Vendas por dia da semana
SELECT
    EXTRACT(DOW FROM data_venda) as dia_semana_num,
    TO_CHAR(data_venda, 'Day') as dia_semana,
    COUNT(*) as total_vendas,
    SUM(valor_total) as valor_total
FROM lab_workspace.vendas
GROUP BY dia_semana_num, dia_semana
ORDER BY dia_semana_num;

-- Análise de fim de semana vs. dia de semana
SELECT
    CASE
        WHEN EXTRACT(DOW FROM data_venda) IN (0, 6) THEN 'Fim de
Semana'
        ELSE 'Dia de Semana'
    END as tipo_dia,

```

```

    COUNT(*) as total_vendas,
    ROUND(AVG(valor_total), 2) as ticket_medio,
    SUM(valor_total) as valor_total
FROM lab_workspace.vendas
GROUP BY tipo_dia;

```

Parte 3: Funções de Janela (Window Functions)

3.1 Rankings e Partições

```

-- Ranking de vendedores por valor total
SELECT
    v.nome as vendedor,
    COUNT(vd.id) as total_vendas,
    SUM(vd.valor_total) as valor_total,
    RANK() OVER (ORDER BY SUM(vd.valor_total) DESC) as ranking
FROM lab_workspace.vendas vd
JOIN lab_workspace.vendedores v ON vd.vendedor_id = v.id
GROUP BY v.nome
ORDER BY ranking;

-- Top 3 produtos por loja
WITH produtos_por_loja AS (
    SELECT
        l.nome as loja,
        p.nome as produto,
        SUM(iv.quantidade) as quantidade_total,
        RANK() OVER (PARTITION BY l.id ORDER BY SUM(iv.quantidade)
DESC) as ranking
    FROM lab_workspace.itens_venda iv
    JOIN lab_workspace.vendas v ON iv.venda_id = v.id
    JOIN lab_workspace.lojas l ON v.loja_id = l.id
    JOIN lab_workspace.produtos p ON iv.produto_id = p.id
    GROUP BY l.id, l.nome, p.nome
)
SELECT loja, produto, quantidade_total, ranking
FROM produtos_por_loja

```

```

WHERE ranking <= 3
ORDER BY loja, ranking;

-- Percentual de vendas por vendedor dentro de cada loja
SELECT
    l.nome as loja,
    v.nome as vendedor,
    COUNT(vd.id) as total_vendas,
    SUM(vd.valor_total) as valor_total,
    ROUND(
        SUM(vd.valor_total) * 100.0 /
        SUM(SUM(vd.valor_total)) OVER (PARTITION BY l.id),
        2) as percentual_loja
FROM lab_workspace.vendas vd
JOIN lab_workspace.vendedores v ON vd.vendedor_id = v.id
JOIN lab_workspace.lojas l ON vd.loja_id = l.id
GROUP BY l.id, l.nome, v.nome
ORDER BY l.nome, percentual_loja DESC;

```

3.2 Análises Cumulativas

```

-- Vendas cumulativas por mês
SELECT
    TO_CHAR(data_venda, 'YYYY-MM') as mes,
    SUM(valor_total) as valor_mensal,
    SUM(SUM(valor_total)) OVER (ORDER BY TO_CHAR(data_venda,
    'YYYY-MM')) as valor_acumulado
FROM lab_workspace.vendas
GROUP BY mes
ORDER BY mes;

-- Média móvel de 7 dias
SELECT
    data_venda,
    SUM(valor_total) as valor_diario,
    ROUND(
        AVG(SUM(valor_total)) OVER (

```

```

        ORDER BY data_venda
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ),
2) as media_movel_7dias
FROM lab_workspace.vendas
GROUP BY data_venda
ORDER BY data_venda;

-- Análise de crescimento percentual
SELECT
    TO_CHAR(data_venda, 'YYYY-MM') as mes,
    SUM(valor_total) as valor_total,
    LAG(SUM(valor_total), 1) OVER (ORDER BY TO_CHAR(data_venda,
'YYYY-MM')) as valor_mes_anterior,
    ROUND(
        (SUM(valor_total) - LAG(SUM(valor_total), 1) OVER (ORDER
BY TO_CHAR(data_venda, 'YYYY-MM'))) /
        NULLIF(LAG(SUM(valor_total), 1) OVER (ORDER BY
TO_CHAR(data_venda, 'YYYY-MM'))), 0) * 100,
    2) as crescimento_percentual
FROM lab_workspace.vendas
GROUP BY mes
ORDER BY mes;

```

Parte 4: Relatórios Dinâmicos

4.1 Pivotamento de Dados

```

-- Vendas por loja e método de pagamento (pivot)
SELECT
    l.nome as loja,
    SUM(CASE WHEN v.metodo_pagamento = 'Crédito' THEN
v.valor_total ELSE 0 END) as credito,
    SUM(CASE WHEN v.metodo_pagamento = 'Débito' THEN v.valor_total
ELSE 0 END) as debito,
    SUM(CASE WHEN v.metodo_pagamento = 'Dinheiro' THEN
v.valor_total ELSE 0 END) as dinheiro,

```

```

        SUM(CASE WHEN v.metodo_pagamento = 'Pix' THEN v.valor_total
ELSE 0 END) as pix,
        SUM(CASE WHEN v.metodo_pagamento = 'Boleto' THEN v.valor_total
ELSE 0 END) as boleto,
        SUM(v.valor_total) as total
FROM lab_workspace.vendas v
JOIN lab_workspace.lojas l ON v.loja_id = l.id
GROUP BY l.nome
ORDER BY total DESC;

-- Vendas por categoria de produto e mês
SELECT
    p.categoria,
    SUM(CASE WHEN TO_CHAR(v.data_venda, 'MM') = '01' THEN
iv.quantidade ELSE 0 END) as jan,
    SUM(CASE WHEN TO_CHAR(v.data_venda, 'MM') = '02' THEN
iv.quantidade ELSE 0 END) as fev,
    SUM(CASE WHEN TO_CHAR(v.data_venda, 'MM') = '03' THEN
iv.quantidade ELSE 0 END) as mar,
    SUM(CASE WHEN TO_CHAR(v.data_venda, 'MM') = '04' THEN
iv.quantidade ELSE 0 END) as abr,
    SUM(CASE WHEN TO_CHAR(v.data_venda, 'MM') = '05' THEN
iv.quantidade ELSE 0 END) as mai,
    SUM(CASE WHEN TO_CHAR(v.data_venda, 'MM') = '06' THEN
iv.quantidade ELSE 0 END) as jun,
    SUM(iv.quantidade) as total
FROM lab_workspace.itens_venda iv
JOIN lab_workspace.vendas v ON iv.venda_id = v.id
JOIN lab_workspace.produtos p ON iv.producto_id = p.id
GROUP BY p.categoria
ORDER BY total DESC;

```

4.2 Análise de Correlação

```

-- Correlação entre tamanho da loja e vendas
SELECT
    l.nome as loja,

```

```

l.tamanho_m2,
COUNT(v.id) as total_vendas,
SUM(v.valor_total) as valor_total,
ROUND(SUM(v.valor_total) / l.tamanho_m2, 2) as venda_por_m2
FROM lab_workspace.vendas v
JOIN lab_workspace.lojas l ON v.loja_id = l.id
GROUP BY l.nome, l.tamanho_m2
ORDER BY venda_por_m2 DESC;

-- Análise de vendas por tempo de experiência do vendedor
SELECT
    v.nome as vendedor,
    EXTRACT(YEAR FROM AGE(CURRENT_DATE, v.data_contratacao)) as anos_experiencia,
    COUNT(vd.id) as total_vendas,
    SUM(vd.valor_total) as valor_total,
    ROUND(AVG(vd.valor_total), 2) as ticket_medio
FROM lab_workspace.vendas vd
JOIN lab_workspace.vendedores v ON vd.vendedor_id = v.id
GROUP BY v.nome, anos_experiencia
ORDER BY anos_experiencia DESC;

-- Correlação entre métodos de pagamento e valor médio
SELECT
    metodo_pagamento,
    COUNT(*) as total_vendas,
    ROUND(AVG(valor_total), 2) as valor_medio,
    ROUND(AVG(parcelas), 2) as media_parcelas
FROM lab_workspace.vendas
GROUP BY metodo_pagamento
ORDER BY valor_medio DESC;

```

4.3 Análise de Segmentação

```

-- Segmentação de clientes por valor
WITH vendas_por_cliente AS (
    SELECT

```

```

        c.id,
        c.nome,
        COUNT(v.id) as total_compras,
        SUM(v.valor_total) as valor_total,
        MAX(v.data_venda) as ultima_compra
    FROM lab_workspace.vendas v
    JOIN lab_workspace.clientes c ON v.cliente_id = c.id
    GROUP BY c.id, c.nome
)
SELECT
    nome as cliente,
    total_compras,
    valor_total,
    ultima_compra,
    CASE
        WHEN valor_total > 5000 THEN 'Premium'
        WHEN valor_total > 2000 THEN 'Regular'
        ELSE 'Básico'
    END as segmento,
    CASE
        WHEN CURRENT_DATE - ultima_compra <= 30 THEN 'Ativo'
        WHEN CURRENT_DATE - ultima_compra <= 90 THEN 'Em risco'
        ELSE 'Inativo'
    END as status_atividade
FROM vendas_por_cliente
ORDER BY valor_total DESC;

-- Análise de produtos por margem de contribuição
SELECT
    p.nome as produto,
    p.categoria,
    SUM(iv.quantidade) as quantidade_vendida,
    SUM(iv.quantidade * iv.preco_unitario) as receita_total,
    ROUND(
        SUM(iv.quantidade * iv.preco_unitario) /
        NULLIF(SUM(iv.quantidade), 0),
        2) as preco_medio_venda,
    p.preco as preco_atual,

```

```

ROUND(
    (p.preco - (SUM(iv.quantidade * iv.preco_unitario) /
NULLIF(SUM(iv.quantidade), 0))) /
    p.preco * 100,
    2) as variacao_percentual
FROM lab_workspace.itens_venda iv
JOIN lab_workspace.produtos p ON iv.producto_id = p.id
GROUP BY p.nome, p.categoria, p.preco
ORDER BY quantidade_vendida DESC;

```

Parte 5: Análise Avançada

5.1 Detecção de Anomalias

```

-- Identificação de vendas atípicas (outliers)
WITH estatisticas_vendas AS (
    SELECT
        AVG(valor_total) as media,
        STDDEV(valor_total) as desvio_padrao
    FROM lab_workspace.vendas
)
SELECT
    v.id,
    v.data_venda,
    c.nome as cliente,
    vd.nome as vendedor,
    v.valor_total,
    ROUND((v.valor_total - e.media) / e.desvio_padrao, 2) as
z_score
FROM lab_workspace.vendas v
JOIN lab_workspace.clientes c ON v.cliente_id = c.id
JOIN lab_workspace.vendedores vd ON v.vendedor_id = vd.id
CROSS JOIN estatisticas_vendas e
WHERE ABS((v.valor_total - e.media) / e.desvio_padrao) > 2
ORDER BY ABS((v.valor_total - e.media) / e.desvio_padrao) DESC;

-- Detecção de padrões incomuns de compra

```

```

SELECT
    v.cliente_id,
    c.nome as cliente,
    COUNT(v.id) as total_compras,
    COUNT(DISTINCT v.data_venda) as dias_distintos,
    ROUND(COUNT(v.id)::NUMERIC / COUNT(DISTINCT v.data_venda), 2)
as compras_por_dia,
    SUM(v.valor_total) as valor_total
FROM lab_workspace.vendas v
JOIN lab_workspace.clientes c ON v.cliente_id = c.id
GROUP BY v.cliente_id, c.nome
HAVING COUNT(v.id) > 10 AND COUNT(v.id)::NUMERIC / COUNT(DISTINCT v.data_venda) > 1.5
ORDER BY compras_por_dia DESC;

```

5.2 Análise Preditiva

```

-- Previsão simples de vendas futuras (média móvel)
WITH vendas_diarias AS (
    SELECT
        data_venda,
        SUM(valor_total) as valor_diario
    FROM lab_workspace.vendas
    GROUP BY data_venda
    ORDER BY data_venda
)
SELECT
    data_venda,
    valor_diario,
    ROUND(
        AVG(valor_diario) OVER (
            ORDER BY data_venda
            ROWS BETWEEN 7 PRECEDING AND 1 PRECEDING
        ),
        2) as previsao_media_movel,
    ROUND(
        valor_diario - AVG(valor_diario) OVER (

```

```

        ORDER BY data_venda
        ROWS BETWEEN 7 PRECEDING AND 1 PRECEDING
    ),
2) as erro_previsao
FROM vendas_diarias
WHERE data_venda > (SELECT MIN(data_venda) + 7 FROM
lab_workspace.vendas)
ORDER BY data_venda;

-- Tendência de crescimento por categoria
WITH vendas_mensais AS (
    SELECT
        TO_CHAR(v.data_venda, 'YYYY-MM') as mes,
        p.categoria,
        SUM(iv.quantidade * iv.preco_unitario) as valor_total
    FROM lab_workspace.itens_venda iv
    JOIN lab_workspace.vendas v ON iv.venda_id = v.id
    JOIN lab_workspace.produtos p ON iv.producto_id = p.id
    GROUP BY mes, p.categoria
    ORDER BY mes, p.categoria
)
SELECT
    categoria,
    CORR(
        EXTRACT(EPOCH FROM TO_DATE(mes, 'YYYY-MM')),
        valor_total
    ) as correlacao_temporal,
    CASE
        WHEN CORR(
            EXTRACT(EPOCH FROM TO_DATE(mes, 'YYYY-MM')),
            valor_total
        ) > 0.7 THEN 'Forte crescimento'
        WHEN CORR(
            EXTRACT(EPOCH FROM TO_DATE(mes, 'YYYY-MM')),
            valor_total
        ) > 0.3 THEN 'Crescimento moderado'
        WHEN CORR(
            EXTRACT(EPOCH FROM TO_DATE(mes, 'YYYY-MM')),

```

```

        valor_total
    ) > -0.3 THEN 'Estável'
WHEN CORR(
    EXTRACT(EPOCH FROM TO_DATE(mes, 'YYYY-MM'))),
    valor_total
) > -0.7 THEN 'Declínio moderado'
ELSE 'Forte declínio'
END as tendência
FROM vendas_mensais
GROUP BY categoria
ORDER BY correlacao_temporal DESC;

```

5.3 Análise de Coorte

```

-- Análise de coorte por mês de primeira compra
WITH primeira_compra AS (
    SELECT
        cliente_id,
        MIN(TO_CHAR(data_venda, 'YYYY-MM')) as coorte
    FROM lab_workspace.vendas
    GROUP BY cliente_id
),
compras_mensais AS (
    SELECT
        pc.coorte,
        TO_CHAR(v.data_venda, 'YYYY-MM') as mes,
        COUNT(DISTINCT v.cliente_id) as clientes_ativos,
        SUM(v.valor_total) as valor_total
    FROM lab_workspace.vendas v
    JOIN primeira_compra pc ON v.cliente_id = pc.cliente_id
    GROUP BY pc.coorte, mes
    ORDER BY pc.coorte, mes
)
SELECT
    coorte,
    mes,
    clientes_ativos,
    valor_total

```

```

valor_total,
ROUND(
    valor_total / clientes_ativos,
2) as valor_medio_por_cliente,
ROUND(
    100.0 * clientes_ativos / (
        SELECT COUNT(DISTINCT cliente_id)
        FROM primeira_compra
        WHERE coorte = cm.coorte
    ),
2) as retencao_percentual
FROM compras_mensais cm
ORDER BY coorte, mes;

```

Parte 6: Visualização e Exportação

6.1 Preparação de Dados para Visualização

```

-- Dados para gráfico de barras (vendas por região)
SELECT
    l.regiao,
    COUNT(v.id) as total_vendas,
    SUM(v.valor_total) as valor_total,
    ROUND(AVG(v.valor_total), 2) as ticket_medio
FROM lab_workspace.vendas v
JOIN lab_workspace.lojas l ON v.loja_id = l.id
GROUP BY l.regiao
ORDER BY valor_total DESC;

-- Dados para gráfico de linha (tendência temporal)
SELECT
    TO_CHAR(data_venda, 'YYYY-MM-DD') as data,
    COUNT(*) as total_vendas,
    SUM(valor_total) as valor_total
FROM lab_workspace.vendas
WHERE data_venda >= CURRENT_DATE - INTERVAL '30 days'
GROUP BY data

```

```

ORDER BY data;

-- Dados para gráfico de pizza (distribuição por categoria)
SELECT
    p.categoria,
    COUNT(iv.venda_id) AS total_vendas,
    SUM(iv.quantidade * iv.preco_unitario) AS valor_total,
    ROUND(
        SUM(iv.quantidade * iv.preco_unitario) * 100.0 /
        (SELECT SUM(quantidade * preco_unitario) FROM
lab_workspace.itens_venda),
        2) AS percentual
FROM lab_workspace.itens_venda iv
JOIN lab_workspace.produtos p ON iv.producto_id = p.id
GROUP BY p.categoria
ORDER BY valor_total DESC;

```

6.2 Exportação de Dados

```

-- Exportar resultados para CSV
COPY (
    SELECT
        v.id,
        v.data_venda,
        c.nome AS cliente,
        vd.nome AS vendedor,
        l.nome AS loja,
        v.valor_total,
        v.metodo_pagamento
    FROM lab_workspace.vendas v
    JOIN lab_workspace.clientes c ON v.cliente_id = c.id
    JOIN lab_workspace.vendedores vd ON v.vendedor_id = vd.id
    JOIN lab_workspace.lojas l ON v.loja_id = l.id
    ORDER BY v.data_venda DESC
) TO '/tmp/relatorio_vendas.csv' WITH CSV HEADER;

-- Exportar relatório de desempenho por vendedor

```

```

COPY (
    SELECT
        vd.nome as vendedor,
        l.nome as loja,
        COUNT(v.id) as total_vendas,
        SUM(v.valor_total) as valor_total,
        ROUND(AVG(v.valor_total), 2) as ticket_medio,
        ROUND(
            SUM(v.valor_total) * 100.0 / NULLIF(vd.meta_mensal,
            0),
            2) as percentual_meta
    FROM lab_workspace.vendas v
    JOIN lab_workspace.vendedores vd ON v.vendedor_id = vd.id
    JOIN lab_workspace.lojas l ON v.loja_id = l.id
    WHERE TO_CHAR(v.data_venda, 'YYYY-MM') = TO_CHAR(CURRENT_DATE,
    'YYYY-MM')
    GROUP BY vd.nome, l.nome, vd.meta_mensal
    ORDER BY percentual_meta DESC
) TO '/tmp/desempenho_vendedores.csv' WITH CSV HEADER;

```

Conclusão do Laboratório

```

-- Registrar conclusão do laboratório
UPDATE lab_workspace.lab_progress
SET
    completion_time = CURRENT_TIMESTAMP,
    status = 'COMPLETED',
    notes = 'Laboratório de análise de dados concluído com
    sucesso. Foram realizadas análises de agregação, temporais,
    funções de janela, relatórios dinâmicos e análises avançadas.'
WHERE lab_name = 'Análise de Dados'
AND completion_time IS NULL;

```

Desafios Adicionais

1. Análise de RFM (Recência, Frequência, Monetário)

- Segmenta clientes com base em padrões de compra
- Identifique clientes de alto valor

2. Análise de Cesta de Compras

- Descubra produtos frequentemente comprados juntos
- Identifique oportunidades de cross-selling

3. Previsão de Demanda

- Utilize técnicas de séries temporais
- Projete vendas futuras por categoria

4. Dashboard Interativo

- Crie visualizações dinâmicas
- Implemente filtros e parâmetros

Recursos Adicionais

- PostgreSQL Window Functions (<https://www.postgresql.org/docs/current/tutorial-window.html>)
- PostgreSQL Aggregation (<https://www.postgresql.org/docs/current/functions-aggregate.html>)
- Data Analysis with SQL (<https://mode.com/sql-tutorial/sql-data-analysis/>)
- Advanced SQL for Data Analysis (<https://www.datacamp.com/courses/advanced-sql>)

⚠ **Nota:** Este laboratório fornece uma base sólida para análise de dados com SQL. As técnicas aprendidas aqui podem ser aplicadas em cenários reais de negócios para extrair insights valiosos e apoiar a tomada de decisões baseada em dados.

Laboratório: Transformação de Dados

Objetivo

Desenvolver habilidades de transformação e manipulação de dados utilizando SQL, incluindo conversão de formatos, limpeza, normalização e enriquecimento de dados.

Cenário

Você é um engenheiro de dados em uma empresa que precisa preparar dados brutos para análise, integrando múltiplas fontes e garantindo a qualidade dos dados.

Setup Inicial

```
-- Criar tabelas para dados brutos
CREATE TABLE lab_workspace.dados_brutos_clientes (
    id VARCHAR(20),
    nome_completo TEXT,
    email TEXT,
    telefone VARCHAR(50),
    endereco TEXT,
    data_cadastro VARCHAR(30),
    status VARCHAR(20),
    observacoes TEXT
);

CREATE TABLE lab_workspace.dados_brutos_vendas (
    id_venda VARCHAR(20),
    data_hora VARCHAR(30),
    cliente_id VARCHAR(20),
    itens TEXT,
    valor_total VARCHAR(20),
    desconto VARCHAR(20),
    forma_pagamento TEXT,
```

```

    status_pagamento TEXT,
    status_entrega TEXT
);

CREATE TABLE lab_workspace.dados_brutos_produtos (
    codigo VARCHAR(20),
    nome TEXT,
    descricao TEXT,
    categoria TEXT,
    subcategoria TEXT,
    fornecedor TEXT,
    preco VARCHAR(20),
    estoque VARCHAR(10),
    unidade_medida VARCHAR(10),
    dimensoes TEXT,
    peso VARCHAR(20),
    data_cadastro VARCHAR(30)
);

-- Carregar dados de exemplo com problemas comuns
INSERT INTO lab_workspace.dados_brutos_clientes
VALUES
    ('CLI-001', 'Silva, João Carlos', 'joao.silva@email.com',
    '(11) 98765-4321', 'Rua das Flores, 123 - São Paulo, SP',
    '15/03/2022', 'ATIVO', NULL),
    ('CLI-002', 'Maria Souza', 'maria.souza@email.com',
    '11987654322', 'Av. Paulista, 1000, Apto 501, São Paulo - SP',
    '2022-04-20', 'ativo', 'Cliente VIP'),
    ('CLI-003', 'PEDRO SANTOS', 'pedro@email', '(21)87654323',
    'Rua Ipanema 45, Rio de Janeiro RJ', '05/05/2022', 'INATIVO',
    'Bloqueado por inadimplência'),
    ('CLI-004', 'Ana Paula Oliveira', 'ana.oliveira@email.com',
    '(31) 9876-54324', 'Av. Amazonas, 500 - Belo Horizonte - MG',
    '10/06/2022', 'ATIVO', NULL),
    ('CLI-005', 'Carlos Eduardo da Silva', 'carlos@email.com.br',
    '(41)998765325', 'R. XV de Novembro, 200 - Curitiba PR',
    '2022/07/15', 'PENDENTE', 'Aguardando confirmação de email');

```

```

INSERT INTO lab_workspace.dados_brutos_vendas
VALUES
    ('V-2022-001', '15/04/2022 14:30:25', 'CLI-001', 'PROD-001:2,PROD-003:1', 'R$ 1.299,90', '10%', 'CARTÃO DE CRÉDITO', 'APROVADO', 'ENTREGUE'),
    ('V-2022-002', '2022-05-20 09:45:12', 'CLI-002', 'PROD-002:1', '2499.90', '0', 'PIX', 'APROVADO', 'EM TRANSPORTE'),
    ('V-2022-003', '25/05/2022 18:20:45', 'CLI-001', 'PROD-005:3,PROD-004:1', 'R$1.549,97', 'R$ 100,00', 'BOLETO', 'APROVADO', 'PENDENTE'),
    ('V-2022-004', '30/05/2022 10:15:30', 'CLI-003', 'PROD-001:1', 'R$ 599,90', 'R$ 0,00', 'DÉBITO', 'APROVADO', 'ENTREGUE'),
    ('V-2022-005', '2022-06-10T08:45:12Z', 'CLI-002', 'PROD-003:2', '1400', '5%', 'CRÉDITO', 'APROVADO', 'ENTREGUE');

INSERT INTO lab_workspace.dados_brutos_produtos
VALUES
    ('PROD-001', 'Smartphone XYZ', 'Smartphone com 128GB de memória', 'Eletrônicos', 'Celulares', 'Tech Inc.', 'R$ 599,90', '15', 'un', '15x7x1 cm', '180g', '10/01/2022'),
    ('PROD-002', 'Notebook Ultra', 'Notebook com processador i7', 'Eletrônicos', 'Computadores', 'Tech Inc.', '2499.90', '8', 'un', NULL, '1,5 kg', '2022-01-15'),
    ('PROD-003', 'Fone de Ouvido Bluetooth', 'Fone sem fio com cancelamento de ruído', 'Eletrônicos', 'Acessórios', 'AudioTech', 'R$699,95', '25', 'un', '18x18x5 cm', '200 g', '20/01/2022'),
    ('PROD-004', 'Mouse Gamer', 'Mouse com 6 botões e LED RGB', 'Eletrônicos', 'Acessórios', 'GamerTech', 'R$ 149,99', '30', 'un', '12x7x4 cm', '100g', '25/01/2022'),
    ('PROD-005', 'Teclado Mecânico', 'Teclado com switches blue', 'Eletrônicos', 'Acessórios', 'GamerTech', 'R$ 299,99', '20', 'un', '44x14x4 cm', '850 g', '2022/01/30');

-- Criar tabela de progresso do laboratório
CREATE TABLE IF NOT EXISTS lab_workspace.lab_progress (
    lab_name VARCHAR(100) PRIMARY KEY,
    start_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    completion_time TIMESTAMP,

```

```

    status VARCHAR(20) DEFAULT 'IN_PROGRESS',
    notes TEXT
);

-- Registrar início do laboratório
INSERT INTO lab_workspace.lab_progress (lab_name)
VALUES ('Transformação de Dados');

```

Parte 1: Limpeza e Padronização

1.1 Padronização de Nomes

```

-- Criar tabela normalizada para clientes
CREATE TABLE lab_workspace.clientes (
    id VARCHAR(20) PRIMARY KEY,
    nome_primeiro VARCHAR(50),
    nome_ultimo VARCHAR(50),
    email VARCHAR(100),
    telefone VARCHAR(20),
    data_cadastro DATE,
    status VARCHAR(20),
    observacoes TEXT
);

-- Padronizar e separar nomes
INSERT INTO lab_workspace.clientes (
    id,
    nome_primeiro,
    nome_ultimo,
    email,
    telefone,
    data_cadastro,
    status,
    observacoes
)
SELECT
    id,

```

```

-- Padronizar e extrair primeiro nome
CASE
    WHEN nome_completo LIKE '%,%' THEN
        TRIM(SPLIT_PART(nome_completo, ',', 2))
    ELSE
        TRIM(SPLIT_PART(nome_completo, ',', 1))
END AS nome_primeiro,

-- Padronizar e extrair último nome
CASE
    WHEN nome_completo LIKE '%,%' THEN
        TRIM(SPLIT_PART(nome_completo, ',', 1))
    ELSE
        CASE
            WHEN ARRAY_LENGTH(STRING_TO_ARRAY(nome_completo,
            '), 1) > 1 THEN
                TRIM(SPLIT_PART(nome_completo, ',',
                ARRAY_LENGTH(STRING_TO_ARRAY(nome_completo, ','), 1)))
            ELSE
                NULL
        END
END AS nome_ultimo,

-- Manter email original por enquanto
email,

-- Padronizar telefone (remover formatação)
REGEXP_REPLACE(telefone, '[^0-9]', '', 'g') AS telefone,

-- Converter data para formato padrão
CASE
    WHEN data_cadastro ~ '^[0-9]{2}/[0-9]{2}/[0-9]{4}$' THEN
        TO_DATE(data_cadastro, 'DD/MM/YYYY')
    WHEN data_cadastro ~ '^[0-9]{4}-[0-9]{2}-[0-9]{2}$' THEN
        TO_DATE(data_cadastro, 'YYYY-MM-DD')
    WHEN data_cadastro ~ '^[0-9]{4}/[0-9]{2}/[0-9]{2}$' THEN
        TO_DATE(data_cadastro, 'YYYY/MM/DD')
    ELSE NULL

```

```

    END AS data_cadastro,

    -- Padronizar status (primeira letra maiúscula, resto
    minúscula)
    INITCAP(LOWER(status)) AS status,

    observacoes
FROM lab_workspace.dados_brutos_clientes;

-- Verificar resultados
SELECT * FROM lab_workspace.clientes;

```

1.2 Validação e Correção de Dados

```

-- Validar e corrigir emails
UPDATE lab_workspace.clientes
SET email =
CASE
    WHEN email NOT LIKE '%@%.%' THEN email || '@dominio.com'
    ELSE email
END
WHERE email NOT LIKE '%@%.%';

-- Validar e padronizar telefones
UPDATE lab_workspace.clientes
SET telefone =
CASE
    WHEN LENGTH(telefone) = 10 THEN
        SUBSTRING(telefone, 1, 2) || '9' ||
        SUBSTRING(telefone, 3)
    ELSE telefone
END
WHERE LENGTH(telefone) = 10;

-- Verificar registros com problemas
SELECT id, email, telefone
FROM lab_workspace.clientes

```

WHERE

```
email !~ '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$' OR  
telefone !~ '^+[0-9]{10,11}$';
```

1.3 Normalização de Endereços

```
-- Criar tabela normalizada para endereços  
CREATE TABLE lab_workspace.enderecos (  
    id SERIAL PRIMARY KEY,  
    cliente_id VARCHAR(20) REFERENCES lab_workspace.clientes(id),  
    logradouro VARCHAR(100),  
    numero VARCHAR(20),  
    complemento VARCHAR(50),  
    bairro VARCHAR(50),  
    cidade VARCHAR(50),  
    estado CHAR(2),  
    cep VARCHAR(8)  
);  
  
-- Extraír e normalizar componentes de endereço  
INSERT INTO lab_workspace.enderecos (  
    cliente_id,  
    logradouro,  
    numero,  
    complemento,  
    cidade,  
    estado  
)  
SELECT  
    id AS cliente_id,  
    -- Extraír logradouro (até o número)  
    REGEXP_REPLACE(  
        SPLIT_PART(  
            SPLIT_PART(endereco, ' ', ' ', 1),  
            ' - ',  
            1  
        ),
```

```

' [0-9]+$',  

'  

) AS logradouro,  
  

-- Extrair número  

REGEXP_REPLACE(  

    SPLIT_PART(  

        SPLIT_PART(endereco, ',', ', ', 1),  

        ' - ',  

        1  

    ),  

    '^.*?([0-9]+)$',  

    '\1'  

) AS numero,  
  

-- Extrair complemento (se existir)  

CASE  

    WHEN endereco LIKE '%Apto%' THEN  

        REGEXP_REPLACE(  

            SPLIT_PART(endereco, ',', ', ', 2),  

            '.*Apto ([0-9]+).*',  

            'Apto \1'  

        )  

    ELSE NULL  

END AS complemento,  
  

-- Extrair cidade  

TRIM(SPLIT_PART(  

    SPLIT_PART(endereco, ' - ', ', ', 2),  

    ' - ',  

    1  

)) AS cidade,  
  

-- Extrair estado  

TRIM(REGEXP_REPLACE(  

    SPLIT_PART(endereco, ' - ', ', ', 2),  

    '.*([A-Z]{2})$',  

    '\1'
)

```

```

)) AS estado
FROM lab_workspace.dados_brutos_clientes;

-- Verificar resultados
SELECT c.id, c.nome_primeiro, c.nome_ultimo, e.logradouro,
e.numero, e.cidade, e.estado
FROM lab_workspace.clientes c
JOIN lab_workspace.enderecos e ON c.id = e.cliente_id;

```

Parte 2: Conversão de Tipos e Formatos

2.1 Normalização de Produtos

```

-- Criar tabela normalizada para produtos
CREATE TABLE lab_workspace.produtos (
    id VARCHAR(20) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    descricao TEXT,
    categoria VARCHAR(50),
    subcategoria VARCHAR(50),
    fornecedor VARCHAR(50),
    preco DECIMAL(10,2) NOT NULL,
    estoque INTEGER NOT NULL,
    unidade_medida VARCHAR(10),
    peso DECIMAL(10,2),
    peso_unidade VARCHAR(5),
    data_cadastro DATE
);

-- Converter e normalizar dados de produtos
INSERT INTO lab_workspace.produtos (
    id,
    nome,
    descricao,
    categoria,
    subcategoria,
    fornecedor,

```

```

    preco,
    estoque,
    unidade_medida,
    peso,
    peso_unidade,
    data_cadastro
)
SELECT
    codigo AS id,
    nome,
    descricao,
    categoria,
    subcategoria,
    fornecedor,
    -- Converter preço para decimal
    CASE
        WHEN preco LIKE 'R$%' THEN
            REPLACE(REPLACE(REPLACE(preco, 'R$', ''), ',', '.'), ',', '')
        ELSE
            REPLACE(preco, ',', '.')::DECIMAL(10, 2)
    END AS preco,
    -- Converter estoque para inteiro
    estoque::INTEGER,
    unidade_medida,
    -- Extrair valor numérico do peso
    CASE
        WHEN peso LIKE '%kg%' THEN
            REPLACE(REPLACE(peso, 'kg', ''), ',', '.', '')::DECIMAL(10, 2) * 1000
        WHEN peso LIKE '%g%' THEN
            REPLACE(REPLACE(peso, 'g', ''), ',', '.', '')::DECIMAL(10, 2)
        ELSE NULL
    END AS peso
)

```

```

    END AS peso,

    -- Extrair unidade do peso
    'g' AS peso_unidade,

    -- Converter data para formato padrão
CASE
    WHEN data_cadastro ~ '^[0-9]{2}/[0-9]{2}/[0-9]{4}$' THEN
        TO_DATE(data_cadastro, 'DD/MM/YYYY')
    WHEN data_cadastro ~ '^[0-9]{4}-[0-9]{2}-[0-9]{2}$' THEN
        TO_DATE(data_cadastro, 'YYYY-MM-DD')
    WHEN data_cadastro ~ '^[0-9]{4}/[0-9]{2}/[0-9]{2}$' THEN
        TO_DATE(data_cadastro, 'YYYY/MM/DD')
    ELSE NULL
END AS data_cadastro
FROM lab_workspace.dados_brutos_produtos;

-- Verificar resultados
SELECT id, nome, preco, estoque, peso, data_cadastro
FROM lab_workspace.produtos;

```

2.2 Normalização de Vendas

```

-- Criar tabela normalizada para vendas
CREATE TABLE lab_workspace.vendas (
    id VARCHAR(20) PRIMARY KEY,
    data_hora TIMESTAMP NOT NULL,
    cliente_id VARCHAR(20) REFERENCES lab_workspace.clientes(id),
    valor_total DECIMAL(10,2) NOT NULL,
    desconto DECIMAL(10,2),
    forma_pagamento VARCHAR(50),
    status_pagamento VARCHAR(20),
    status_entrega VARCHAR(20)
);

-- Converter e normalizar dados de vendas
INSERT INTO lab_workspace.vendas (

```

```

        id,
        data_hora,
        cliente_id,
        valor_total,
        desconto,
        forma_pagamento,
        status_pagamento,
        status_entrega
    )
SELECT
    id_venda AS id,

    -- Converter data_hora para timestamp
    CASE
        WHEN data_hora ~ '^[0-9]{2}/[0-9]{2}/[0-9]{4} [0-9]{2}:[0-9]{2}:[0-9]{2}$' THEN
            TO_TIMESTAMP(data_hora, 'DD/MM/YYYY HH24:MI:SS')
        WHEN data_hora ~ '^[0-9]{4}-[0-9]{2}-[0-9]{2} [0-9]{2}:[0-9]{2}:[0-9]{2}$' THEN
            TO_TIMESTAMP(data_hora, 'YYYY-MM-DD HH24:MI:SS')
        ELSE
            TO_TIMESTAMP(data_hora, 'YYYY-MM-DDTHH24:MI:SSZ')
    END AS data_hora,

    cliente_id,

    -- Converter valor_total para decimal
    CASE
        WHEN valor_total LIKE 'R$%' THEN
            REPLACE(REPLACE(REPLACE(valor_total, 'R$', ''), '.', ','), ',', '.', '.')
        ELSE
            REPLACE(valor_total, ',', '.')
    END AS valor_total,

    -- Converter desconto para decimal
    CASE
        WHEN desconto LIKE '%' THEN

```

```

        REPLACE(desconto, '%', '')::DECIMAL(10,2) / 100
    ELSE
        REPLACE(desconto, ',', '.')::DECIMAL(10,2)
END AS desconto,

forma_pagamento,
status_pagamento,
status_entrega
FROM lab_workspace.dados_brutos_vendas;

-- Verificar resultados
SELECT id, data_hora, cliente_id, valor_total, desconto,
forma_pagamento, status_pagamento, status_entrega
FROM lab_workspace.vendas;

```

Parte 3: Enriquecimento de Dados

3.1 Adicionar Informações de Geolocalização

```

-- Criar tabela para armazenar informações de geolocalização
CREATE TABLE lab_workspace.geolocalizacao (
    cidade VARCHAR(50),
    estado CHAR(2),
    pais VARCHAR(50),
    latitude DECIMAL(9,6),
    longitude DECIMAL(9,6)
);

-- Inserir dados de geolocalização (exemplo)
INSERT INTO lab_workspace.geolocalizacao (cidade, estado, pais,
latitude, longitude)
VALUES
    ('São Paulo', 'SP', 'Brasil', -23.550520, -46.633309),
    ('Rio de Janeiro', 'RJ', 'Brasil', -22.906847, -43.172897),
    ('Belo Horizonte', 'MG', 'Brasil', -19.815733, -43.954234),
    ('Curitiba', 'PR', 'Brasil', -25.428408, -49.273388);

```

```
-- Adicionar colunas para geolocalização na tabela de endereços
ALTER TABLE lab_workspace.enderecos
ADD COLUMN latitude DECIMAL(9, 6),
ADD COLUMN longitude DECIMAL(9, 6);

-- Atualizar endereços com informações de geolocalização
UPDATE lab_workspace.enderecos e
SET latitude = g.latitude,
    longitude = g.longitude
FROM lab_workspace.geolocalizacao g
WHERE e.cidade = g.cidade AND e.estado = g.estado;

-- Verificar resultados
SELECT c.id, c.nome_primeiro, c.nome_ultimo, e.logradouro,
e.numero, e.cidade, e.estado, e.latitude, e.longitude
FROM lab_workspace.clientes c
JOIN lab_workspace.enderecos e ON c.id = e.cliente_id;
```

3.2 Adicionar Informações de Produtos

```
-- Criar tabela para armazenar informações adicionais de produtos
CREATE TABLE lab_workspace.produtos_info (
    produto_id VARCHAR(20) PRIMARY KEY,
    marca VARCHAR(50),
    modelo VARCHAR(50),
    garantia VARCHAR(50),
    fabricante VARCHAR(50)
);

-- Inserir dados de informações de produtos (exemplo)
INSERT INTO lab_workspace.produtos_info (produto_id, marca,
modelo, garantia, fabricante)
VALUES
    ('PROD-001', 'Tech Inc.', 'XYZ', '1 ano', 'Tech Inc.'),
    ('PROD-002', 'Tech Inc.', 'Ultra', '2 anos', 'Tech Inc.'),
    ('PROD-003', 'AudioTech', 'Bluetooth', '1 ano', 'AudioTech'),
    ('PROD-004', 'GamerTech', 'Gamer', '1 ano', 'GamerTech'),
```

```

('PROD-005', 'GamerTech', 'Mechanical', '2 anos',
'GamerTech');

-- Adicionar colunas para informações de produtos na tabela de
produtos
ALTER TABLE lab_workspace.produtos
ADD COLUMN marca VARCHAR(50),
ADD COLUMN modelo VARCHAR(50),
ADD COLUMN garantia VARCHAR(50),
ADD COLUMN fabricante VARCHAR(50);

-- Atualizar produtos com informações adicionais
UPDATE lab_workspace.produtos p
SET marca = pi.marca,
    modelo = pi.modelo,
    garantia = pi.garantia,
    fabricante = pi.fabricante
FROM lab_workspace.produtos_info pi
WHERE p.id = pi.producto_id;

-- Verificar resultados
SELECT p.id, p.nome, p.preco, p.estoque, p.peso, p.data_cadastro,
p.marca, p.modelo, p.garantia, p.fabricante
FROM lab_workspace.produtos p;

```

Parte 4: Análise e Visualização

4.1 Análise de Vendas

```

-- Calcular o total de vendas por cliente
SELECT c.id, c.nome_primeiro, c.nome_ultimo, SUM(v.valor_total) AS
total_vendas
FROM lab_workspace.clientes c
JOIN lab_workspace.vendas v ON c.id = v.cliente_id
GROUP BY c.id, c.nome_primeiro, c.nome_ultimo
ORDER BY total_vendas DESC;

```

```

-- Calcular o total de vendas por produto
SELECT p.id, p.nome, SUM(v.valor_total) AS total_vendas
FROM lab_workspace.produtos p
JOIN lab_workspace.vendas v ON p.id = v.itens
GROUP BY p.id, p.nome
ORDER BY total_vendas DESC;

-- Calcular o total de vendas por mês
SELECT TO_CHAR(v.data_hora, 'YYYY-MM') AS mes, SUM(v.valor_total)
AS total_vendas
FROM lab_workspace.vendas v
GROUP BY TO_CHAR(v.data_hora, 'YYYY-MM')
ORDER BY mes;

```

4.2 Visualização de Dados

Para visualizar os dados, você pode usar ferramentas como Tableau, Power BI ou Google Data Studio. Aqui estão algumas ideias de visualizações:

1. Gráfico de barras para o total de vendas por cliente.
2. Gráfico de pizza para a distribuição de vendas por produto.
3. Gráfico de linha para o total de vendas ao longo do tempo.
4. Mapa para visualizar a distribuição geográfica dos clientes.

Lembre-se de que a visualização de dados é uma parte importante da análise e pode ajudar a identificar tendências e padrões nos dados.

Conclusão

Neste laboratório, você aprendeu a transformar e manipular dados brutos utilizando SQL, incluindo limpeza, normalização, conversão de formatos e enriquecimento de dados. Você também explorou técnicas de análise e visualização de dados para obter insights valiosos.

Próximos Passos

Para aprofundar seus conhecimentos, você pode:

1. Explorar mais técnicas de manipulação de dados em SQL.
2. Aprender sobre ferramentas de análise e visualização de dados.
3. Trabalhar com conjuntos de dados maiores e mais complexos.
4. Experimentar diferentes abordagens de limpeza e normalização de dados.

Boa sorte! Se tiver alguma dúvida, sinta-se à vontade para perguntar.

Laboratório: Integração de Dados

Objetivo

Desenvolver habilidades de integração de dados utilizando SQL, incluindo importação, exportação, sincronização entre tabelas, migração e validação de integridade.

Cenário

Você é um engenheiro de dados responsável por integrar dados de diferentes fontes em um sistema unificado, garantindo consistência e qualidade durante todo o processo.

Setup Inicial

```
-- Criar tabelas para o laboratório
CREATE TABLE lab_workspace.sistema_legado_clientes (
    cliente_id INTEGER PRIMARY KEY,
    nome VARCHAR(100),
    email VARCHAR(100),
    telefone VARCHAR(20),
    data_cadastro DATE,
    ultima_compra DATE,
    status VARCHAR(20)
);

CREATE TABLE lab_workspace.sistema_legado_produtos (
    produto_id INTEGER PRIMARY KEY,
    nome VARCHAR(100),
    categoria VARCHAR(50),
    preco DECIMAL(10, 2),
    estoque INTEGER,
    fornecedor VARCHAR(100)
);

CREATE TABLE lab_workspace.sistema_legado_vendas (
    venda_id INTEGER PRIMARY KEY,
```

```

cliente_id INTEGER,
data_venda DATE,
valor_total DECIMAL(10,2),
status VARCHAR(20)
);

CREATE TABLE lab_workspace.sistema_legado_itens_venda (
venda_id INTEGER,
produto_id INTEGER,
quantidade INTEGER,
preco_unitario DECIMAL(10,2),
PRIMARY KEY (venda_id, produto_id)
);

-- Criar tabelas do novo sistema
CREATE TABLE lab_workspace.clientes_novo (
id SERIAL PRIMARY KEY,
codigo_legado INTEGER UNIQUE,
nome VARCHAR(100) NOT NULL,
email VARCHAR(100) UNIQUE,
telefone VARCHAR(20),
data_cadastro DATE,
ultima_atualizacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
status VARCHAR(20) DEFAULT 'Ativo',
segmento VARCHAR(50),
limite_credito DECIMAL(10,2) DEFAULT 1000.00
);

CREATE TABLE lab_workspace.produtos_novo (
id SERIAL PRIMARY KEY,
codigo_legado INTEGER UNIQUE,
nome VARCHAR(100) NOT NULL,
descricao TEXT,
categoria VARCHAR(50),
subcategoria VARCHAR(50),
preco DECIMAL(10,2) NOT NULL,
custo DECIMAL(10,2),
margem_lucro DECIMAL(5,2),

```

```

estoque INTEGER DEFAULT 0,
estoque_minimo INTEGER DEFAULT 5,
fornecedor VARCHAR(100),
data_cadastro DATE,
ultima_atualizacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
status VARCHAR(20) DEFAULT 'Ativo'
);

CREATE TABLE lab_workspace.vendas_novo (
    id SERIAL PRIMARY KEY,
    codigo_legado INTEGER UNIQUE,
    cliente_id INTEGER REFERENCES lab_workspace.clientes_novo(id),
    data_venda TIMESTAMP NOT NULL,
    valor_subtotal DECIMAL(10, 2) NOT NULL,
    valor_desconto DECIMAL(10, 2) DEFAULT 0,
    valor_frete DECIMAL(10, 2) DEFAULT 0,
    valor_total DECIMAL(10, 2) NOT NULL,
    forma_pagamento VARCHAR(50),
    status VARCHAR(20) DEFAULT 'Processando',
    data_atualizacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE lab_workspace.itens_venda_novo (
    venda_id INTEGER REFERENCES lab_workspace.vendas_novo(id),
    produto_id INTEGER REFERENCES lab_workspace.produtos_novo(id),
    quantidade INTEGER NOT NULL CHECK (quantidade > 0),
    preco_unitario DECIMAL(10, 2) NOT NULL,
    valor_desconto DECIMAL(10, 2) DEFAULT 0,
    valor_total DECIMAL(10, 2) NOT NULL,
    PRIMARY KEY (venda_id, produto_id)
);

-- Inserir dados de exemplo no sistema legado
INSERT INTO lab_workspace.sistema_legado_clientes
VALUES
    (1001, 'João Silva', 'joao.silva@email.com', '11987654321',
    '2020-01-15', '2022-05-10', 'Ativo'),
    (1002, 'Maria Santos', 'maria.santos@email.com',

```

```

'21987654321', '2020-02-20', '2022-06-15', 'Ativo'),
(1003, 'Pedro Oliveira', 'pedro.oliveira@email.com',
'31987654321', '2020-03-25', '2022-04-05', 'Inativo'),
(1004, 'Ana Costa', 'ana.costa@email.com', '41987654321',
'2020-04-30', '2022-06-20', 'Ativo'),
(1005, 'Carlos Souza', 'carlos.souza@email.com',
'51987654321', '2020-05-05', '2022-03-10', 'Inativo');

INSERT INTO lab_workspace.sistema_legado_produtos
VALUES
(2001, 'Notebook Basic', 'Informática', 2499.90, 15, 'Tech
Distribuidora'),
(2002, 'Smartphone X', 'Eletrônicos', 1899.90, 25, 'Mobile
Imports'),
(2003, 'Mouse Sem Fio', 'Informática', 89.90, 50, 'Tech
Distribuidora'),
(2004, 'Teclado Mecânico', 'Informática', 299.90, 30, 'Tech
Distribuidora'),
(2005, 'Monitor 24"', 'Informática', 899.90, 10, 'Visual
Displays');

INSERT INTO lab_workspace.sistema_legado_vendas
VALUES
(3001, 1001, '2022-05-10', 2589.80, 'Concluída'),
(3002, 1002, '2022-06-15', 1899.90, 'Concluída'),
(3003, 1003, '2022-04-05', 389.80, 'Concluída'),
(3004, 1004, '2022-06-20', 3399.80, 'Concluída'),
(3005, 1005, '2022-03-10', 89.90, 'Cancelada');

INSERT INTO lab_workspace.sistema_legado_itens_venda
VALUES
(3001, 2001, 1, 2499.90),
(3001, 2003, 1, 89.90),
(3002, 2002, 1, 1899.90),
(3003, 2003, 1, 89.90),
(3003, 2004, 1, 299.90),
(3004, 2001, 1, 2499.90),
(3004, 2005, 1, 899.90),

```

```

(3005, 2003, 1, 89.90);

-- Registrar início do laboratório
INSERT INTO lab_workspace.lab_progress (lab_name)
VALUES ('Integração de Dados');

```

Parte 1: Importação de Dados

1.1 Migração de Clientes

```

-- Migrar clientes do sistema legado para o novo sistema
INSERT INTO lab_workspace.clientes_novo (
    codigo_legado,
    nome,
    email,
    telefone,
    data_cadastro,
    ultima_atualizacao,
    status,
    segmento,
    limite_credito
)
SELECT
    cliente_id,
    nome,
    email,
    telefone,
    data_cadastro,
    CURRENT_TIMESTAMP,
    status,
    CASE
        WHEN ultima_compra >= CURRENT_DATE - INTERVAL '3 months'
        THEN 'Frequente'
        WHEN ultima_compra >= CURRENT_DATE - INTERVAL '6 months'
        THEN 'Regular'
        ELSE 'Ocasional'
    END AS segmento,

```

```

CASE
    WHEN ultima_compra >= CURRENT_DATE - INTERVAL '3 months'
THEN 2000.00
    WHEN ultima_compra >= CURRENT_DATE - INTERVAL '6 months'
THEN 1500.00
    ELSE 1000.00
END AS limite_credito
FROM lab_workspace.sistema_legado_clientes;

-- Verificar resultados
SELECT * FROM lab_workspace.clientes_novo;

```

1.2 Migração de Produtos

```

-- Migrar produtos do sistema legado para o novo sistema
INSERT INTO lab_workspace.produtos_novo (
    codigo_legado,
    nome,
    categoria,
    preco,
    custo,
    margem_lucro,
    estoque,
    estoque_minimo,
    fornecedor,
    data_cadastro,
    ultima_atualizacao,
    status
)
SELECT
    produto_id,
    nome,
    categoria,
    preco,
    preco * 0.7 AS custo, -- Estimativa de custo (70% do preço)
    0.3 AS margem_lucro, -- Margem de lucro estimada (30%)
    estoque,

```

```

CASE
    WHEN categoria = 'Informática' THEN 10
    WHEN categoria = 'Eletrônicos' THEN 5
    ELSE 3
END AS estoque_minimo,
fornecedor,
CURRENT_DATE - INTERVAL '1 year' AS dataCadastro, -- Data
estimada
CURRENT_TIMESTAMP,
'Ativo' AS status
FROM lab_workspace.sistema_legado_produtos;

-- Adicionar descrições aos produtos
UPDATE lab_workspace.produtos_novo
SET descricao =
CASE
    WHEN nome LIKE '%Notebook%' THEN 'Notebook para uso
pessoal e profissional'
    WHEN nome LIKE '%Smartphone%' THEN 'Smartphone com câmera
de alta resolução'
    WHEN nome LIKE '%Mouse%' THEN 'Mouse ergonômico com
conexão sem fio'
    WHEN nome LIKE '%Teclado%' THEN 'Teclado mecânico com
iluminação RGB'
    WHEN nome LIKE '%Monitor%' THEN 'Monitor LED com alta
definição'
    ELSE 'Produto eletrônico de alta qualidade'
END;

-- Adicionar subcategorias
UPDATE lab_workspace.produtos_novo
SET subcategoria =
CASE
    WHEN nome LIKE '%Notebook%' THEN 'Laptops'
    WHEN nome LIKE '%Smartphone%' THEN 'Celulares'
    WHEN nome LIKE '%Mouse%' THEN 'Periféricos'
    WHEN nome LIKE '%Teclado%' THEN 'Periféricos'
    WHEN nome LIKE '%Monitor%' THEN 'Monitores'

```

```

    ELSE 'Outros'
END;

-- Verificar resultados
SELECT * FROM lab_workspace.produtos_novo;

```

1.3 Migração de Vendas

```

-- Migrar vendas do sistema legado para o novo sistema
INSERT INTO lab_workspace.vendas_novo (
    codigo_legado,
    cliente_id,
    data_venda,
    valor_subtotal,
    valor_desconto,
    valor_frete,
    valor_total,
    forma_pagamento,
    status,
    data_atualizacao
)
SELECT
    v.venda_id,
    c.id AS cliente_id,
    v.data_venda::TIMESTAMP,
    v.valor_total AS valor_subtotal,
    0 AS valor_desconto,
    0 AS valor_frete,
    v.valor_total,
    'Cartão de Crédito' AS forma_pagamento, -- Valor padrão
    CASE
        WHEN v.status = 'Concluída' THEN 'Finalizada'
        WHEN v.status = 'Cancelada' THEN 'Cancelada'
        ELSE 'Processando'
    END AS status,
    CURRENT_TIMESTAMP
FROM lab_workspace.sistema_legado_vendas v

```

```

JOIN lab_workspace.clientes_novo c ON v.cliente_id = 
c.codigo_legado;

-- Migrar itens de venda
INSERT INTO lab_workspace.itens_venda_novo (
    venda_id,
    produto_id,
    quantidade,
    preco_unitario,
    valor_desconto,
    valor_total
)
SELECT
    v.id AS venda_id,
    p.id AS produto_id,
    i.quantidade,
    i.preco_unitario,
    0 AS valor_desconto,
    i.quantidade * i.preco_unitario AS valor_total
FROM lab_workspace.sistema_legado_itens_venda i
JOIN lab_workspace.vendas_novo v ON i.venda_id = v.codigo_legado
JOIN lab_workspace.produtos_novo p ON i.produto_id = 
p.codigo_legado;

-- Verificar resultados
SELECT * FROM lab_workspace.vendas_novo;
SELECT * FROM lab_workspace.itens_venda_novo;

```

Parte 2: Sincronização de Dados

2.1 Criação de Tabelas de Staging

```

-- Criar tabelas de staging para novos dados
CREATE TABLE lab_workspace.staging_clientes (
    cliente_id INTEGER PRIMARY KEY,
    nome VARCHAR(100),
    email VARCHAR(100),

```

```

    telefone VARCHAR(20),
    data_cadastro DATE,
    ultima_compra DATE,
    status VARCHAR(20),
    observacoes TEXT
);

CREATE TABLE lab_workspace.staging_produtos (
    produto_id INTEGER PRIMARY KEY,
    nome VARCHAR(100),
    categoria VARCHAR(50),
    subcategoria VARCHAR(50),
    preco DECIMAL(10,2),
    custo DECIMAL(10,2),
    estoque INTEGER,
    fornecedor VARCHAR(100),
    status VARCHAR(20)
);

-- Inserir dados de exemplo nas tabelas de staging
INSERT INTO lab_workspace.staging_clientes
VALUES
    (1001, 'João Silva', 'joao.silva.novo@email.com',
    '11987654321', '2020-01-15', '2022-07-05', 'Ativo', 'Cliente
atualizado'),
    (1006, 'Fernanda Lima', 'fernanda.lima@email.com',
    '61987654321', '2022-07-01', '2022-07-02', 'Ativo', 'Novo
cliente'),
    (1007, 'Roberto Alves', 'roberto.alves@email.com',
    '71987654321', '2022-07-02', '2022-07-03', 'Ativo', 'Novo
cliente');

INSERT INTO lab_workspace.staging_produtos
VALUES
    (2001, 'Notebook Basic Plus', 'Informática', 'Laptops',
    2699.90, 1889.93, 10, 'Tech Distribuidora', 'Ativo'),
    (2006, 'Headphone Bluetooth', 'Eletrônicos', 'Áudio', 199.90,
    139.93, 40, 'Sound Solutions', 'Ativo'),

```

```
(2007, 'Webcam HD', 'Informática', 'Periféricos', 149.90,  
104.93, 30, 'Visual Tech', 'Ativo');
```

2.2 Sincronização de Clientes

```
-- Sincronizar clientes (inserir novos e atualizar existentes)  
WITH clientes_atualizados AS (  
    -- Atualizar clientes existentes  
    UPDATE lab_workspace.clientes_novo c  
    SET  
        nome = s.nome,  
        email = s.email,  
        telefone = s.telefone,  
        status = s.status,  
        ultima_atualizacao = CURRENT_TIMESTAMP,  
        segmento = CASE  
            WHEN s.ultima_compra >= CURRENT_DATE - INTERVAL '3  
months' THEN 'Frequente'  
            WHEN s.ultima_compra >= CURRENT_DATE - INTERVAL '6  
months' THEN 'Regular'  
            ELSE 'Ocasional'  
        END  
    FROM lab_workspace.staging_clientes s  
    WHERE c.codigo_legado = s.cliente_id  
    RETURNING c.codigo_legado  
)  
-- Inserir novos clientes  
INSERT INTO lab_workspace.clientes_novo (  
    codigo_legado,  
    nome,  
    email,  
    telefone,  
    data_cadastro,  
    ultima_atualizacao,  
    status,  
    segmento,  
    limite_credito
```

```

)
SELECT
    cliente_id,
    nome,
    email,
    telefone,
    data_cadastro,
    CURRENT_TIMESTAMP,
    status,
    CASE
        WHEN ultima_compra >= CURRENT_DATE - INTERVAL '3 months'
    THEN 'Frequente'
        WHEN ultima_compra >= CURRENT_DATE - INTERVAL '6 months'
    THEN 'Regular'
        ELSE 'Ocasional'
    END AS segmento,
    CASE
        WHEN ultima_compra >= CURRENT_DATE - INTERVAL '3 months'
    THEN 2000.00
        WHEN ultima_compra >= CURRENT_DATE - INTERVAL '6 months'
    THEN 1500.00
        ELSE 1000.00
    END AS limite_credito
FROM lab_workspace.staging_clientes s
WHERE NOT EXISTS (
    SELECT 1 FROM clientes_atualizados ca
    WHERE ca.codigo_legado = s.cliente_id
);
-- Verificar resultados
SELECT * FROM lab_workspace.clientes_novo ORDER BY id;

```

2.3 Sincronização de Produtos

```

-- Sincronizar produtos (inserir novos e atualizar existentes)
WITH produtos_atualizados AS (
    -- Atualizar produtos existentes

```

```

UPDATE lab_workspace.produtos_novo p
SET
    nome = s.nome,
    categoria = s.categoria,
    subcategoria = s.subcategoria,
    preco = s.preco,
    custo = s.custo,
    estoque = s.estoque,
    fornecedor = s.fornecedor,
    status = s.status,
    ultima_atualizacao = CURRENT_TIMESTAMP
FROM lab_workspace.staging_produtos s
WHERE p.codigo_legado = s.producto_id
RETURNING p.codigo_legado
)
-- Inserir novos produtos
INSERT INTO lab_workspace.produtos_novo (
    codigo_legado,
    nome,
    categoria,
    subcategoria,
    preco,
    custo,
    margem_lucro,
    estoque,
    estoque_minimo,
    fornecedor,
    data_cadastro,
    ultima_atualizacao,
    status
)
SELECT
    produto_id,
    nome,
    categoria,
    subcategoria,
    preco,
    custo,

```

```

    (preco - custo) / preco AS margem_lucro,
    estoque,
    CASE
        WHEN categoria = 'Informática' THEN 10
        WHEN categoria = 'Eletrônicos' THEN 5
        ELSE 3
    END AS estoque_minimo,
    fornecedor,
    CURRENT_DATE AS data_cadastro,
    CURRENT_TIMESTAMP,
    status
FROM lab_workspace.staging_produtos s
WHERE NOT EXISTS (
    SELECT 1 FROM produtos_atualizados pa
    WHERE pa.codigo_legado = s.producto_id
);
-- Verificar resultados
SELECT * FROM lab_workspace.produtos_novo ORDER BY id;

```

Parte 3: Validação e Correção de Dados

3.1 Verificação de Integridade Referencial

```

-- Verificar integridade referencial entre vendas e clientes
SELECT v.id, v.codigo_legado, v.cliente_id
FROM lab_workspace.vendas_novo v
LEFT JOIN lab_workspace.clientes_novo c ON v.cliente_id = c.id
WHERE c.id IS NULL;

-- Verificar integridade referencial entre itens de venda e
produtos
SELECT i.venda_id, i.producto_id
FROM lab_workspace.itens_venda_novo i
LEFT JOIN lab_workspace.produtos_novo p ON i.producto_id = p.id
WHERE p.id IS NULL;

```

```
-- Verificar integridade referencial entre itens de venda e vendas
SELECT i.venda_id, i.product_id
FROM lab_workspace.itens_venda_novo i
LEFT JOIN lab_workspace.vendas_novo v ON i.venda_id = v.id
WHERE v.id IS NULL;
```

3.2 Correção de Inconsistências

```
-- Atualizar valores totais das vendas com base nos itens
UPDATE lab_workspace.vendas_novo v
SET
    valor_subtotal = subquery.subtotal,
    valor_total = subquery.subtotal
FROM (
    SELECT
        venda_id,
        SUM(quantidade * preco_unitario) AS subtotal
    FROM lab_workspace.itens_venda_novo
    GROUP BY venda_id
) AS subquery
WHERE v.id = subquery.venda_id
AND (v.valor_subtotal <> subquery.subtotal OR v.valor_total <>
subquery.subtotal);

-- Verificar resultados
SELECT
    v.id,
    v.valor_subtotal,
    v.valor_total,
    (SELECT SUM(quantidade * preco_unitario) FROM
lab_workspace.itens_venda_novo WHERE venda_id = v.id) AS calculado
FROM lab_workspace.vendas_novo v;
```

3.3 Enriquecimento de Dados

```

-- Adicionar informações de forma de pagamento com base em padrões
UPDATE lab_workspace.vendas_novo
SET forma_pagamento =
CASE
    WHEN id % 3 = 0 THEN 'Cartão de Crédito'
    WHEN id % 3 = 1 THEN 'Boleto Bancário'
    WHEN id % 3 = 2 THEN 'PIX'
END
WHERE forma_pagamento = 'Cartão de Crédito';

-- Adicionar valor de frete com base no valor total
UPDATE lab_workspace.vendas_novo
SET
    valor_frete =
CASE
    WHEN valor_total < 100 THEN 15.00
    WHEN valor_total < 500 THEN 25.00
    WHEN valor_total < 1000 THEN 35.00
    ELSE 0.00 -- Frete grátis para compras acima de 1000
END,
    valor_total = valor_total +
CASE
    WHEN valor_total < 100 THEN 15.00
    WHEN valor_total < 500 THEN 25.00
    WHEN valor_total < 1000 THEN 35.00
    ELSE 0.00
END;

-- Verificar resultados
SELECT id, valor_subtotal, valor_frete, valor_total,
forma_pagamento
FROM lab_workspace.vendas_novo;

```

Parte 4: Exportação e Relatórios

4.1 Criação de Visões para Relatórios

```

-- Visão de resumo de vendas por cliente
CREATE OR REPLACE VIEW lab_workspace.vw_vendas_por_cliente AS
SELECT
    c.id AS cliente_id,
    c.nome,
    c.segmento,
    COUNT(v.id) AS total_pedidos,
    SUM(v.valor_total) AS valor_total_compras,
    MAX(v.data_venda) AS ultima_compra,
    MIN(v.data_venda) AS primeira_compra
FROM lab_workspace.clientes_novo c
LEFT JOIN lab_workspace.vendas_novo v ON c.id = v.cliente_id
GROUP BY c.id, c.nome, c.segmento
ORDER BY valor_total_compras DESC NULLS LAST;

-- Visão de produtos mais vendidos
CREATE OR REPLACE VIEW lab_workspace.vw_produtos_mais_vendidos AS
SELECT
    p.id AS produto_id,
    p.nome,
    p.categoria,
    p.subcategoria,
    SUM(i.quantidade) AS quantidade_vendida,
    SUM(i.quantidade * i.preco_unitario) AS valor_total_vendas,
    COUNT(DISTINCT i.venda_id) AS numero_vendas,
    p.estoque AS estoque_atual,
    CASE
        WHEN p.estoque = 0 THEN 'Sem estoque'
        WHEN p.estoque < p.estoque_minimo THEN 'Estoque baixo'
        ELSE 'Estoque normal'
    END AS status_estoque
FROM lab_workspace.produtos_novo p
LEFT JOIN lab_workspace.itens_venda_novo i ON p.id = i.produto_id
GROUP BY p.id, p.nome, p.categoria, p.subcategoria, p.estoque,
p.estoque_minimo
ORDER BY quantidade_vendida DESC NULLS LAST;

```

```
-- Consultar visões
SELECT * FROM lab_workspace.vw_vendas_por_cliente;
SELECT * FROM lab_workspace.vw_produtos_mais_vendidos;
```

4.2 Exportação para Tabelas de Relatórios

```
-- Criar tabela de relatório de vendas mensais
CREATE TABLE lab_workspace.relatorio_vendas_mensais (
    ano INTEGER,
    mes INTEGER,
    categoria VARCHAR(50),
    subcategoria VARCHAR(50),
    total_vendas INTEGER,
    valor_total DECIMAL(12, 2),
    ticket_medio DECIMAL(12, 2),
    data_geracao TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Preencher tabela de relatório
INSERT INTO lab_workspace.relatorio_vendas_mensais (
    ano,
    mes,
    categoria,
    subcategoria,
    total_vendas,
    valor_total,
    ticket_medio
)
SELECT
    EXTRACT(YEAR FROM v.data_venda) AS ano,
    EXTRACT(MONTH FROM v.data_venda) AS mes,
    p.categoria,
    p.subcategoria,
    COUNT(DISTINCT v.id) AS total_vendas,
    SUM(i.quantidade * i.preco_unitario) AS valor_total,
    SUM(i.quantidade * i.preco_unitario) / COUNT(DISTINCT v.id) AS
    ticket_medio
```

```

FROM lab_workspace.vendas_novo v
JOIN lab_workspace.itens_venda_novo i ON v.id = i.venda_id
JOIN lab_workspace.produtos_novo p ON i.producto_id = p.id
GROUP BY
    EXTRACT(YEAR FROM v.data_venda),
    EXTRACT(MONTH FROM v.data_venda),
    p.categoria,
    p.subcategoria
ORDER BY
    ano, mes, categoria, subcategoria;

-- Verificar resultados
SELECT * FROM lab_workspace.relatorio_vendas_mensais;

```

4.3 Exportação para Formatos Externos

```

-- Preparar dados para exportação em CSV (simulado com uma
consulta)
SELECT
    c.id,
    c.nome,
    c.email,
    c.telefone,
    c.segmento,
    COUNT(v.id) AS total_pedidos,
    SUM(v.valor_total) AS valor_total,
    MAX(v.data_venda) AS ultima_compra
FROM lab_workspace.clientes_novo c
LEFT JOIN lab_workspace.vendas_novo v ON c.id = v.cliente_id
GROUP BY c.id, c.nome, c.email, c.telefone, c.segmento
ORDER BY c.id;

-- Preparar dados para exportação em JSON (simulado com uma
consulta)
SELECT
    json_build_object(
        'id', p.id,

```

```

    'nome', p.nome,
    'categoria', p.categoria,
    'subcategoria', p.subcategoria,
    'preco', p.preco,
    'estoque', p.estoque,
    'vendas', (
        SELECT json_agg(
            json_build_object(
                'venda_id', i.venda_id,
                'quantidade', i.quantidade,
                'valor', i.preco_unitario
            )
        )
    )
    FROM lab_workspace.itens_venda_novo i
    WHERE i.producto_id = p.id
)
) AS produto_json
FROM lab_workspace.produtos_novo p
ORDER BY p.id;

```

Parte 5: Manutenção e Limpeza

5.1 Arquivamento de Dados

```

-- Criar tabela de arquivamento para vendas antigas
CREATE TABLE lab_workspace.vendas_arquivo (
    id INTEGER,
    codigo_legado INTEGER,
    cliente_id INTEGER,
    data_venda TIMESTAMP,
    valor_subtotal DECIMAL(10, 2),
    valor_desconto DECIMAL(10, 2),
    valor_frete DECIMAL(10, 2),
    valor_total DECIMAL(10, 2),
    forma_pagamento VARCHAR(50),
    status VARCHAR(20),
    data_arquivamento TIMESTAMP DEFAULT CURRENT_TIMESTAMP

```

```
);

-- Mover vendas antigas para arquivamento
WITH vendas_arquivadas AS (
    DELETE FROM lab_workspace.vendas_novo
    WHERE data_venda < CURRENT_DATE - INTERVAL '1 year'
    RETURNING *
)
INSERT INTO lab_workspace.vendas_arquivo (
    id,
    codigo_legado,
    cliente_id,
    data_venda,
    valor_subtotal,
    valor_desconto,
    valor_frete,
    valor_total,
    forma_pagamento,
    status
)
SELECT
    id,
    codigo_legado,
    cliente_id,
    data_venda,
    valor_subtotal,
    valor_desconto,
    valor_frete,
    valor_total,
    forma_pagamento,
    status
FROM vendas_arquivadas;

-- Verificar resultados
SELECT * FROM lab_workspace.vendas_arquivo;
```

5.2 Limpeza de Dados Temporários

```
-- Limpar tabelas de staging após sincronização
TRUNCATE TABLE lab_workspace.staging_clientes;
TRUNCATE TABLE lab_workspace.staging_produtos;

-- Verificar se as tabelas estão vazias
SELECT COUNT(*) FROM lab_workspace.staging_clientes;
SELECT COUNT(*) FROM lab_workspace.staging_produtos;
```

5.3 Atualização de Estatísticas

```
-- Atualizar estatísticas para melhorar performance de consultas
ANALYZE lab_workspace.clientes_novo;
ANALYZE lab_workspace.produtos_novo;
ANALYZE lab_workspace.vendas_novo;
ANALYZE lab_workspace.itens_venda_novo;

-- Registrar conclusão do laboratório
UPDATE lab_workspace.lab_progress
SET
    completion_time = CURRENT_TIMESTAMP,
    status = 'COMPLETED',
    notes = 'Laboratório de integração de dados concluído com sucesso. Foram realizadas operações de migração, sincronização, validação, exportação e manutenção de dados.'
WHERE lab_name = 'Integração de Dados'
AND completion_time IS NULL;
```

Desafios Adicionais

1. Implementar um processo de ETL completo

- Extrair dados de múltiplas fontes
- Transformar e normalizar os dados

- Carregar em um data warehouse

2. Criar um sistema de detecção de anomalias

- Identificar valores atípicos
- Detectar padrões suspeitos
- Gerar alertas automáticos

3. Desenvolver um processo de reconciliação de dados

- Comparar dados entre sistemas
- Identificar discrepâncias
- Resolver conflitos automaticamente

Conclusão

Neste laboratório, você praticou técnicas essenciais de integração de dados, incluindo:

- Migração de dados entre sistemas
- Sincronização e atualização de registros
- Validação e correção de inconsistências
- Enriquecimento de dados
- Criação de relatórios e exportação
- Manutenção e arquivamento

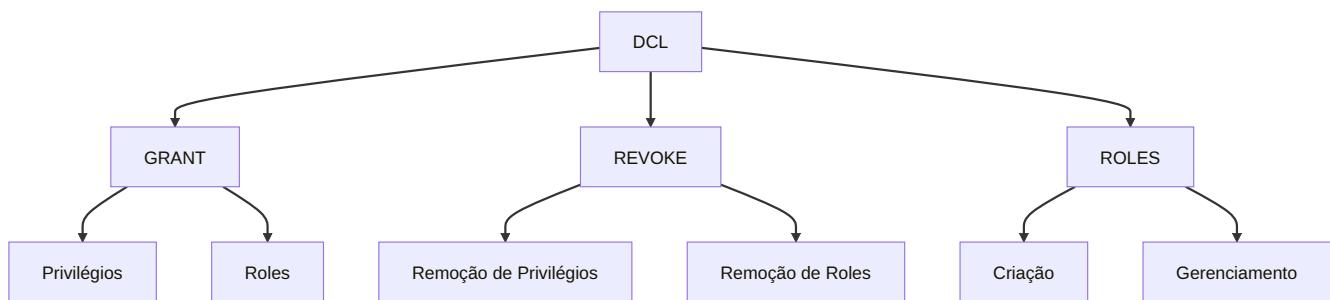
Estas habilidades são fundamentais para profissionais que trabalham com engenharia de dados, ETL, data warehousing e análise de dados.

DCL - Linguagem de Controle de Dados

|| SECURITY_MASTER >> Controle de Acesso a Dados
|| INSTRUTOR: SECURITY_SPECIALIST

Visão Geral

SECURITY_SPECIALIST apresenta: "A Linguagem de Controle de Dados (DCL) é fundamental para gerenciar segurança e acesso em bancos de dados."



Comandos Principais

1. GRANT

PERMISSION_MASTER explica: "Concede privilégios a usuários e roles"

```
-- Privilégios básicos
GRANT SELECT, INSERT ON tabela TO usuario;

-- Todos os privilégios
GRANT ALL PRIVILEGES ON DATABASE banco TO admin;

-- Privilégios específicos
GRANT UPDATE(salario) ON funcionarios TO rh_manager;
```

2. REVOKE

ACCESS_CONTROLLER demonstra: "Remove privilégios concedidos"

```
-- Revogar privilégios específicos  
REVOKE INSERT, UPDATE ON produtos FROM usuario;  
  
-- Revogar todos os privilégios  
REVOKE ALL PRIVILEGES ON ALL TABLES IN SCHEMA public FROM usuario;
```

3. Gerenciamento de ROLES

ROLE_MASTER apresenta: "Organiza privilégios em grupos lógicos"

```
-- Criar role  
CREATE ROLE analistas;  
  
-- Atribuir privilégios à role  
GRANT SELECT ON ALL TABLES IN SCHEMA analytics TO analistas;  
  
-- Atribuir role a usuário  
GRANT analistas TO usuario;
```

Padrões de Segurança

SECURITY_ARCHITECT compartilha padrões essenciais:

1. Princípio do Menor Privilégio

```
-- Criar role com privilégios mínimos  
CREATE ROLE app_read;  
GRANT SELECT ON tabela1, tabela2 TO app_read;  
GRANT USAGE ON SCHEMA public TO app_read;  
  
-- Revogar privilégios desnecessários  
REVOKE ALL ON ALL TABLES IN SCHEMA public FROM PUBLIC;
```

2. Hierarquia de Roles

```
-- Estrutura hierárquica
CREATE ROLE junior_dev;
CREATE ROLE senior_dev;
CREATE ROLE tech_lead;

GRANT junior_dev TO senior_dev;
GRANT senior_dev TO tech_lead;
```

3. Segurança em Nível de Coluna

```
-- Restringir acesso a colunas sensíveis
GRANT SELECT (nome, email) ON usuarios TO suporte;
REVOKE SELECT (senha_hash) ON usuarios FROM suporte;
```

Boas Práticas

SECURITY GUARDIAN compartilha diretrizes:

1. Auditoria

```
-- Criar tabela de auditoria
CREATE TABLE audit_log (
    id SERIAL PRIMARY KEY,
    usuario TEXT,
    acao TEXT,
    tabela TEXT,
    data TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Trigger para auditoria
CREATE TRIGGER audit_changes
AFTER INSERT OR UPDATE OR DELETE ON dados_sensiveis
FOR EACH ROW EXECUTE FUNCTION log_changes();
```

2. Revisão Regular

```
-- Consultar privilégios existentes
SELECT grantee, privilege_type, table_name
FROM information_schema.role_table_grants
WHERE table_schema = 'public';

-- Verificar membros de roles
SELECT rolname, member
FROM pg_roles r
JOIN pg_auth_members m ON r.oid = m.roleid;
```

3. Rotação de Credenciais

```
-- Alterar senha de usuário
ALTER USER aplicacao_user WITH PASSWORD 'novo_password_seguro';

-- Revogar e reconectar sessões
SELECT pg_terminate_backend(pid)
FROM pg_stat_activity
WHERE username = 'aplicacao_user';
```

Cenários Comuns

IMPLEMENTATION_EXPERT apresenta soluções práticas:

1. Ambiente de Desenvolvimento

```
-- Setup inicial
CREATE ROLE dev_team;
GRANT CONNECT ON DATABASE dev_db TO dev_team;
GRANT USAGE ON SCHEMA public TO dev_team;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA
public TO dev_team;
```

2. Ambiente de Produção

```
-- Setup de produção
CREATE ROLE app_service;
```

```
GRANT CONNECT ON DATABASE prod_db TO app_service;
GRANT USAGE ON SCHEMA public TO app_service;
GRANT SELECT, INSERT ON tabela_cliente TO app_service;
GRANT UPDATE (status, ultima_atualizacao) ON tabela_pedido TO
app_service;
```

Troubleshooting

PROBLEM_SOLVER apresenta soluções para problemas comuns:

1. Conflitos de Privilégios

```
-- Verificar privilégios efetivos
SELECT * FROM information_schema.role_table_grants
WHERE grantee = 'usuario_problematico';

-- Resetar privilégios
REVOKE ALL PRIVILEGES ON ALL TABLES IN SCHEMA public FROM
usuario_problematico;
GRANT SELECT ON tabela_especifica TO usuario_problematico;
```

2. Problemas de Acesso

```
-- Diagnosticar problemas de conexão
SELECT * FROM pg_stat_activity
WHERE username = 'usuario_problema';

-- Verificar configurações de autenticação
SELECT rolname, rolcanlogin, rolvaliduntil
FROM pg_roles
WHERE rolname = 'usuario_problema';
```

CHECKLIST SEGURANÇA:	
<input type="checkbox"/> Privilégios mínimos?	
<input type="checkbox"/> Roles apropriadas?	

- | | | |
|--|---|--|
| | <input type="checkbox"/> Auditoria ativa? | |
| | <input type="checkbox"/> Senhas fortes? | |
| | <input type="checkbox"/> Revisão regular? | |

Conclusão

SECURITY_SPECIALIST conclui: "O DCL é crucial para manter a segurança e integridade do banco de dados. Implemente controles de acesso cuidadosamente e revise-os regularmente."



Dica de Segurança: Mantenha um inventário atualizado de todos os privilégios concedidos e revise-o periodicamente para garantir conformidade com as políticas de segurança.

GRANT: Concedendo Privilégios

```
|| PERMISSION_MASTER >> Gerenciamento de Privilégios  
|| INSTRUTOR: GRANT_SPECIALIST
```

Sintaxe Básica

GRANT_SPECIALIST explica: "O comando GRANT é fundamental para controle de acesso!"

```
-- Sintaxe básica  
GRANT privilégio ON objeto TO destinatário;  
  
-- Múltiplos privilégios  
GRANT SELECT, INSERT, UPDATE ON tabela TO usuario;  
  
-- Todos os privilégios  
GRANT ALL PRIVILEGES ON tabela TO admin;
```

Tipos de Privilégios

PRIVILEGE_MASTER apresenta os principais privilégios:

1. Privilégios de Tabela

```
-- Privilégios básicos  
GRANT SELECT ON clientes TO analista;  
GRANT INSERT, UPDATE ON produtos TO vendedor;  
GRANT DELETE ON temp_dados TO admin;  
  
-- Privilégios em colunas específicas  
GRANT SELECT (nome, email) ON usuarios TO suporte;  
GRANT UPDATE (preco, estoque) ON produtos TO gerente;
```

2. Privilégios de Schema

```
-- Acesso ao schema  
GRANT USAGE ON SCHEMA public TO usuario;  
GRANT CREATE ON SCHEMA analytics TO dev_team;  
  
-- Todos os objetos do schema  
GRANT SELECT ON ALL TABLES IN SCHEMA public TO leitor;
```

3. Privilégios de Database

```
-- Privilégios de banco  
GRANT CONNECT ON DATABASE app_db TO app_user;  
GRANT TEMPORARY ON DATABASE temp_db TO etl_user;  
GRANT ALL PRIVILEGES ON DATABASE prod_db TO admin;
```

Cenários Comuns

IMPLEMENTATION_EXPERT demonstra casos práticos:

1. Setup de Aplicação

```
-- Usuário de aplicação  
GRANT CONNECT ON DATABASE app_db TO app_user;  
GRANT USAGE ON SCHEMA public TO app_user;  
GRANT SELECT, INSERT, UPDATE ON tabela_clientes TO app_user;  
GRANT EXECUTE ON FUNCTION proc_negocio TO app_user;
```

2. Setup de Analista

```
-- Analista de dados  
GRANT CONNECT ON DATABASE analytics_db TO analista;  
GRANT USAGE ON SCHEMA relatorios TO analista;  
GRANT SELECT ON ALL TABLES IN SCHEMA relatorios TO analista;  
GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA relatorios TO analista;
```

Boas Práticas

SECURITY_GUARDIAN compartilha diretrizes:

1. Princípio do Menor Privilégio

```
-- Privilégios específicos
GRANT SELECT ON vw_relatorio TO analista;
GRANT INSERT ON log_acesso TO app_user;

-- Evitar privilégios excessivos
REVOKE ALL ON ALL TABLES IN SCHEMA public FROM PUBLIC;
```

2. Gerenciamento via Roles

```
-- Criar e configurar role
CREATE ROLE vendedores;
GRANT SELECT, INSERT ON pedidos TO vendedores;
GRANT UPDATE (status) ON pedidos TO vendedores;

-- Atribuir role a usuários
GRANT vendedores TO usuario1, usuario2;
```

3. Auditoria

```
-- Visualizar privilégios
SELECT grantee, privilege_type, table_name
FROM information_schema.role_table_grants
WHERE table_schema = 'public';

-- Logging de alterações
CREATE TRIGGER audit_grants
AFTER GRANT ON *.*
FOR EACH STATEMENT
EXECUTE FUNCTION log_grant_changes();
```

CHECKLIST DE CONCESSÃO:

- Privilégios mínimos necessários?
- Role apropriada existe?
- Escopo bem definido?
- Auditoria configurada?
- Documentação atualizada?

Troubleshooting

PROBLEM_SOLVER apresenta soluções:

1. Verificação de Privilégios

```
-- Verificar privilégios concedidos
SELECT * FROM information_schema.table_privileges
WHERE grantee = 'usuario';

-- Verificar roles do usuário
SELECT r.rolname
FROM pg_roles r
JOIN pg_auth_members m ON m.roleid = r.oid
WHERE m.member = 'usuario'::regrole;
```

2. Resolução de Problemas

```
-- Revogar e reconceder privilégios
REVOKE ALL PRIVILEGES ON tabela FROM usuario;
GRANT SELECT, INSERT ON tabela TO usuario;

-- Verificar conflitos
SELECT * FROM information_schema.role_table_grants
WHERE table_name = 'tabela_problema';
```

Conclusão

GRANT_SPECIALIST conclui: "GRANT é uma ferramenta poderosa para segurança. Use-a com sabedoria e mantenha documentação clara dos privilégios concedidos."



Dica Final: Revise periodicamente os privilégios concedidos e mantenha um registro de todas as alterações de permissões para auditoria futura.

REVOKE: Revogando Privilégios

```
|| SECURITY_MASTER >> Gerenciamento de Revogações  
|| INSTRUTOR: REVOKE_SPECIALIST
```

Sintaxe Básica

REVOKE_SPECIALIST explica: "O comando REVOKE é essencial para manter a segurança e controle de acesso!"

```
-- Sintaxe básica  
REVOKE privilégio ON objeto FROM destinatário;  
  
-- Múltiplos privilégios  
REVOKE SELECT, INSERT, UPDATE ON tabela FROM usuario;  
  
-- Todos os privilégios  
REVOKE ALL PRIVILEGES ON tabela FROM usuario;
```

Tipos de Revogação

SECURITY_MASTER apresenta os principais cenários:

1. Revogação de Tabela

```
-- Privilégios básicos  
REVOKE SELECT ON clientes FROM analista;  
REVOKE INSERT, UPDATE ON produtos FROM vendedor;  
REVOKE DELETE ON temp_dados FROM usuario;  
  
-- Privilégios em colunas específicas  
REVOKE SELECT (senha, dados_sensíveis) ON usuarios FROM suporte;  
REVOKE UPDATE (salario) ON funcionários FROM gerente;
```

2. Revogação de Schema

```
-- Acesso ao schema  
REVOKE USAGE ON SCHEMA confidencial FROM usuario;  
REVOKE CREATE ON SCHEMA public FROM dev_team;  
  
-- Todos os objetos do schema  
REVOKE SELECT ON ALL TABLES IN SCHEMA dados_sensiveis FROM grupo;
```

3. Revogação de Database

```
-- Privilégios de banco  
REVOKE CONNECT ON DATABASE prod_db FROM test_user;  
REVOKE ALL PRIVILEGES ON DATABASE analytics FROM old_user;  
REVOKE TEMPORARY ON DATABASE temp_db FROM temp_user;
```

Cenários Comuns

IMPLEMENTATION_EXPERT demonstra casos práticos:

1. Desativação de Usuário

```
-- Remoção completa de acessos  
REVOKE ALL PRIVILEGES ON ALL TABLES IN SCHEMA public FROM  
usuario_desativado;  
REVOKE ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public FROM  
usuario_desativado;  
REVOKE ALL PRIVILEGES ON ALL FUNCTIONS IN SCHEMA public FROM  
usuario_desativado;  
REVOKE USAGE ON SCHEMA public FROM usuario_desativado;
```

2. Ajuste de Permissões

```
-- Refinamento de acesso  
REVOKE INSERT, UPDATE, DELETE ON dados_financeiros FROM analista;  
GRANT SELECT ON dados_financeiros TO analista;
```

Boas Práticas

SECURITY GUARDIAN compartilha diretrizes:

1. Revogação Segura

```
-- Verificar antes de revogar
SELECT * FROM information_schema.role_table_grants
WHERE grantee = 'usuario_alvo';

-- Revogar com cautela
BEGIN;
    REVOKE critical_role FROM usuario;
    -- Verificar impacto
    SELECT current_user, session_user;
COMMIT;
```

2. Auditoria de Revogações

```
-- Logging de revogações
CREATE TABLE revoke_audit_log (
    id SERIAL PRIMARY KEY,
    revoked_from VARCHAR(100),
    revoked_privilege VARCHAR(50),
    revoked_object VARCHAR(100),
    revoked_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    revoked_by VARCHAR(100)
);

-- Trigger para auditoria
CREATE TRIGGER log_revokes
AFTER REVOKE ON *.* 
FOR EACH STATEMENT
EXECUTE FUNCTION log_revoke_changes();
```

|| CHECKLIST DE REVOGAÇÃO: ||

- | | |
|----------------------------------|--|
| □ Impacto analisado? | |
| □ Backup de permissões feito? | |
| □ Usuários afetados notificados? | |
| □ Auditoria configurada? | |
| □ Plano de contingência pronto? | |

Troubleshooting

PROBLEM_SOLVER apresenta soluções:

1. Verificação de Impacto

```
-- Analisar dependências
SELECT r.rolname, m.member
FROM pg_roles r
JOIN pg_auth_members m ON r.oid = m.roleid
WHERE r.rolname = 'role_a_revogar';

-- Verificar sessões ativas
SELECT pid, usename, application_name
FROM pg_stat_activity
WHERE usename = 'usuario_alvo';
```

2. Recuperação de Acesso

```
-- Backup de privilégios
CREATE TABLE privilege_backup AS
SELECT * FROM information_schema.role_table_grants
WHERE grantee = 'usuario_alvo';

-- Restauração de privilégios
-- (Gerar scripts de GRANT baseados no backup)
```

Conclusão

REVOKE_SPECIALIST conclui: "REVOKE é uma operação crítica para segurança - use com planejamento e cautela."



Dica Final: Sempre documente as revogações realizadas e mantenha um histórico das alterações de permissões para referência futura.

Gerenciamento de Roles

```
|| ROLE_MASTER >> Gerenciamento de Roles e Grupos  
|| INSTRUTOR: ROLE_SPECIALIST
```

Conceitos Básicos

ROLE_SPECIALIST explica: "Roles são fundamentais para organizar e gerenciar permissões de forma eficiente!"

```
-- Criar role básica  
CREATE ROLE nome_role;  
  
-- Criar role com login  
CREATE ROLE usuario_app WITH  
    LOGIN  
    PASSWORD 'senha_segura'  
    VALID UNTIL '2024-12-31';
```

Tipos de Roles

1. Roles de Aplicação

```
-- Role para aplicação web  
CREATE ROLE app_web WITH LOGIN;  
GRANT CONNECT ON DATABASE app_db TO app_web;  
GRANT USAGE ON SCHEMA public TO app_web;  
GRANT SELECT, INSERT, UPDATE ON tabela_usuarios TO app_web;
```

2. Roles de Grupo

```
-- Role para equipe de analistas  
CREATE ROLE analistas;
```

```
GRANT SELECT ON ALL TABLES IN SCHEMA analytics TO analistas;
GRANT USAGE ON SCHEMA analytics TO analistas;

-- Atribuir usuários ao grupo
GRANT analistas TO analista1, analista2, analista3;
```

Hierarquia de Roles

HIERARCHY_MASTER demonstra a estruturação:

```
-- Estrutura hierárquica
CREATE ROLE junior_dev;
CREATE ROLE senior_dev;
CREATE ROLE tech_lead;

-- Estabelecer hierarquia
GRANT junior_dev TO senior_dev;
GRANT senior_dev TO tech_lead;

-- Configurar permissões por nível
GRANT SELECT ON app_tables TO junior_dev;
GRANT INSERT, UPDATE ON app_tables TO senior_dev;
GRANT ALL PRIVILEGES ON app_tables TO tech_lead;
```

Gerenciamento de Permissões

1. Atribuição de Privilégios

```
-- Privilégios básicos
GRANT SELECT, INSERT ON tabela TO role_name;

-- Privilégios administrativos
GRANT CREATE ON DATABASE app_db TO admin_role;
GRANT USAGE, CREATE ON SCHEMA public TO dev_role;
```

2. Revogação de Privilégios

```
-- Revogar privilégios  
REVOKE INSERT, UPDATE ON dados_sensitiveis FROM role_name;  
  
-- Revogar membership  
REVOKE analistas FROM ex_analista;
```

Boas Práticas

SECURITY_EXPERT compartilha diretrizes:

1. Organização

```
-- Nomenclatura consistente  
CREATE ROLE app_read; -- Apenas leitura  
CREATE ROLE app_write; -- Leitura e escrita  
CREATE ROLE app_admin; -- Administração  
  
-- Documentação  
COMMENT ON ROLE app_read IS 'Role para acesso somente leitura';
```

2. Auditoria

```
-- Visualizar membros de role  
SELECT r.rolname, m.member, u.username  
FROM pg_roles r  
LEFT JOIN pg_auth_members m ON r.oid = m.roleid  
LEFT JOIN pg_user u ON m.member = u.usename;  
  
-- Verificar privilégios  
SELECT * FROM information_schema.role_table_grants  
WHERE grantee = 'role_name';
```

CHECKLIST DE ROLES:

- Nomenclatura padronizada?
- Hierarquia bem definida?

- | | |
|---|--|
| <input type="checkbox"/> Privilégios mínimos necessários? | |
| <input type="checkbox"/> Documentação atualizada? | |
| <input type="checkbox"/> Auditoria configurada? | |

Cenários Comuns

IMPLEMENTATION_EXPERT apresenta soluções práticas:

1. Setup de Desenvolvimento

```
-- Role para desenvolvedores
CREATE ROLE dev_team;
GRANT CONNECT ON DATABASE dev_db TO dev_team;
GRANT USAGE, CREATE ON SCHEMA public TO dev_team;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO dev_team;
```

2. Setup de Produção

```
-- Role para aplicação em produção
CREATE ROLE prod_app WITH LOGIN;
GRANT CONNECT ON DATABASE prod_db TO prod_app;
GRANT USAGE ON SCHEMA public TO prod_app;
GRANT SELECT, INSERT ON tabela_clientes TO prod_app;
GRANT EXECUTE ON FUNCTION proc_negocio TO prod_app;
```

Troubleshooting

PROBLEM_SOLVER apresenta soluções:

1. Diagnóstico

```
-- Verificar conflitos
SELECT r.rolname,
       r.rolsuper,
       r.rolinherit,
       r.rolcreaterole,
```

```

r.rolcreatedb
FROM pg_roles r
WHERE r.rolname = 'role_problema';

-- Verificar sessões ativas
SELECT pid, usename, application_name
FROM pg_stat_activity
WHERE usename IN (
    SELECT m.member::regrole::text
    FROM pg_roles r
    JOIN pg_auth_members m ON r.oid = m.roleid
    WHERE r.rolname = 'role_problema'
);

```

2. Manutenção

```

-- Limpar roles não utilizadas
SELECT r.rolname
FROM pg_roles r
LEFT JOIN pg_auth_members m ON r.oid = m.roleid
WHERE m.roleid IS NULL
AND NOT r.rolcanlogin;

-- Atualizar expiração
ALTER ROLE usuario_temporario
VALID UNTIL '2024-06-30';

```

Conclusão

ROLE_SPECIALIST conclui: "Um bom gerenciamento de roles é fundamental para segurança e organização do banco de dados."



Dica Final: Revise regularmente as roles e suas permissões, mantendo documentação atualizada e removendo acessos desnecessários.

Exercícios de DCL (Data Control Language)

Visão Geral

Este módulo contém uma série progressiva de exercícios para praticar comandos DCL em SQL. Os exercícios estão organizados em três níveis de dificuldade:

Nível Básico

- Criação e gerenciamento de usuários
- Concessão de privilégios simples
- Revogação de privilégios
- Operações fundamentais de DCL

Nível Intermediário

- Gerenciamento de roles
- Hierarquias de privilégios
- Privilégios em nível de coluna
- Técnicas de segurança intermediárias

Nível Avançado

- Implementação de políticas de segurança
- Auditoria de privilégios
- Cenários empresariais de controle de acesso
- Migração de permissões

Estrutura dos Exercícios

Cada exercício segue o formato:

1. Descrição do problema
2. Requisitos específicos
3. Dicas de implementação
4. Solução de referência
5. Critérios de avaliação

Ambiente de Prática

```
-- Database de teste
CREATE DATABASE segurança_db;

-- Schema para isolamento
CREATE SCHEMA aplicacao;
CREATE SCHEMA relatorios;
CREATE SCHEMA administracao;

-- Tabelas de exemplo
CREATE TABLE aplicacao.clientes (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    cpf VARCHAR(14) UNIQUE,
    limite_credito DECIMAL(10, 2),
    data_cadastro DATE DEFAULT CURRENT_DATE
);

CREATE TABLE aplicacao.produtos (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    descricao TEXT,
```

```

    preco DECIMAL(10,2) CHECK (preco > 0),
    estoque INTEGER DEFAULT 0
);

CREATE TABLE aplicacao_pedidos (
    id SERIAL PRIMARY KEY,
    cliente_id INTEGER REFERENCES aplicacao_clientes(id),
    data_pedido TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    valor_total DECIMAL(10,2),
    status VARCHAR(20) DEFAULT 'Pendente'
);

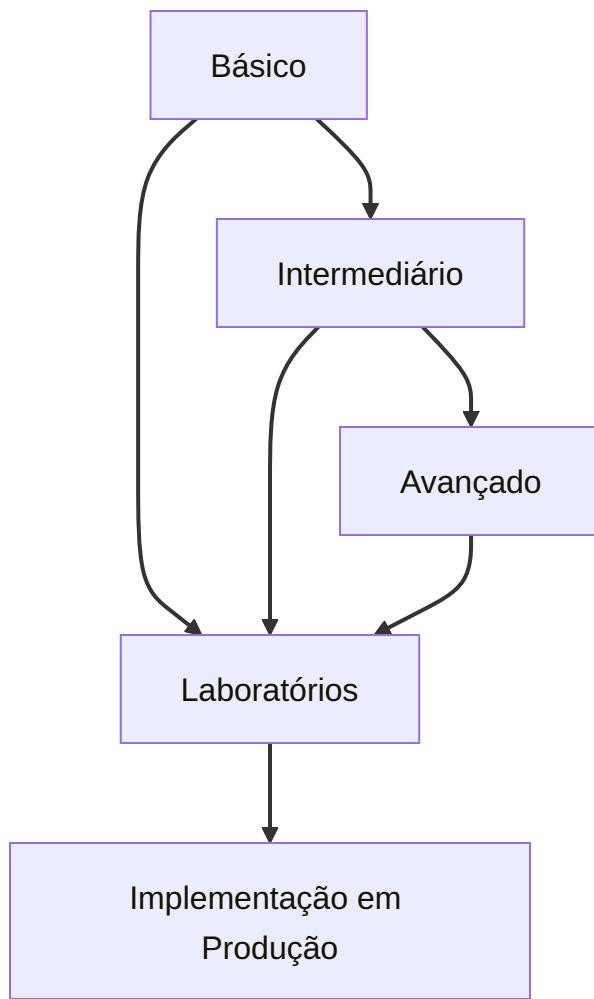
CREATE TABLE relatorios_vendas_mensais (
    mes DATE PRIMARY KEY,
    total_vendas DECIMAL(12,2),
    total_pedidos INTEGER,
    ticket_medio DECIMAL(10,2)
);

CREATE TABLE administracao_usuarios (
    username VARCHAR(50) PRIMARY KEY,
    departamento VARCHAR(50),
    cargo VARCHAR(50),
    data_criacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE administracao_log_acessos (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50),
    tabela_acessada VARCHAR(100),
    tipo_operacao VARCHAR(20),
    data_acesso TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

```

Fluxo de Estudo Recomendado



Preparação do Ambiente

Para obter o máximo proveito destes exercícios, recomendamos:

1. Configurar um ambiente de banco de dados

- PostgreSQL 12+ ou MySQL 8+
- Cliente SQL (DBeaver, pgAdmin, MySQL Workbench)
- Scripts de inicialização fornecidos

2. Usuários de teste

```
-- Criar usuários para testes
CREATE USER app_user WITH PASSWORD 'app123';
CREATE USER relatorio_user WITH PASSWORD 'rel123';
```

```
CREATE USER admin_user WITH PASSWORD 'adm123';
CREATE USER dev_user WITH PASSWORD 'dev123';
```

3. Ferramentas recomendadas

- Editor de SQL com highlight de sintaxe
- Ferramenta para visualização de privilégios
- Ambiente para documentar suas soluções

Navegação do Conteúdo

- Exercícios Básicos ([Exercícios Básicos de DCL](#))
- Exercícios Intermediários ([Exercícios Intermediários de DCL](#))
- Exercícios Avançados ([Exercícios Avançados de DCL](#))

Dicas de Estudo

BOAS PRÁTICAS DE SEGURANÇA:

- ✓ Privilégio mínimo necessário
- ✓ Usar roles para agrupar permissões
- ✓ Revisar permissões regularmente
- ✓ Documentar políticas de acesso
- ✓ Implementar auditoria de acessos

Supporte e Recursos

- Fórum de discussão
- Documentação oficial do SGBD
- Exemplos práticos adicionais

- Soluções comentadas

Próximos Passos

Após completar os exercícios de DCL, recomendamos:

1. Laboratórios práticos

- Aplicação em cenários realistas
- Integração com outros conceitos

2. Projetos integrados

- Implementação de políticas de segurança
- Auditoria e conformidade

3. Tópicos relacionados

- Segurança de banco de dados
- Criptografia de dados
- Conformidade com regulamentações (LGPD, GDPR)

Feedback e Avaliação

Para cada conjunto de exercícios, você pode:

- Verificar suas soluções contra as referências
- Compartilhar abordagens alternativas
- Discutir implicações de segurança
- Solicitar revisão por pares

Exercícios Básicos de DCL

Exercício 1: Concessão de Privilégios Básicos

Descrição

Pratique a concessão de privilégios básicos para usuários em diferentes tabelas.

Requisitos

- Conceder privilégios de SELECT
- Conceder privilégios de INSERT/UPDATE
- Conceder privilégios de DELETE
- Verificar privilégios concedidos

Solução

```
-- Conceder privilégio de SELECT
GRANT SELECT ON aplicacao.produtos TO app_user;

-- Conceder privilégios de INSERT e UPDATE
GRANT INSERT, UPDATE ON aplicacao_pedidos TO app_user;

-- Conceder privilégio de DELETE com restrição
GRANT DELETE ON aplicacao_pedidos TO admin_user;

-- Verificar privilégios concedidos
SELECT grantee, table_schema, table_name, privilege_type
FROM information_schema.table_privileges
WHERE grantee = 'app_user';
```

Exercício 2: Revogação de Privilégios

Descrição

Pratique a revogação de privilégios previamente concedidos.

Requisitos

- Revogar privilégios específicos
- Revogar todos os privilégios
- Revogar privilégios de múltiplos objetos
- Verificar após revogação

Solução

```
-- Revogar privilégio específico  
REVOKE INSERT ON aplicacao.pedidos FROM app_user;  
  
-- Revogar todos os privilégios de uma tabela  
REVOKE ALL PRIVILEGES ON aplicacao.clientes FROM app_user;  
  
-- Revogar privilégios de múltiplas tabelas  
REVOKE SELECT ON aplicacao.produtos, aplicacao_pedidos FROM  
app_user;  
  
-- Verificar após revogação  
SELECT grantee, table_schema, table_name, privilege_type  
FROM information_schema.table_privileges  
WHERE grantee = 'app_user';
```

Exercício 3: Privilégios em Nível de Schema

Descrição

Pratique a concessão e revogação de privilégios em nível de schema.

Requisitos

- Conceder USAGE em schema

- Conceder privilégios em todas as tabelas
- Conceder privilégios em tabelas futuras
- Revogar privilégios em nível de schema

Solução

```
-- Conceder USAGE em schema
GRANT USAGE ON SCHEMA relatorios TO relatorio_user;

-- Conceder privilégios em todas as tabelas existentes
GRANT SELECT ON ALL TABLES IN SCHEMA relatorios TO relatorio_user;

-- Conceder privilégios em tabelas futuras
ALTER DEFAULT PRIVILEGES IN SCHEMA relatorios
GRANT SELECT ON TABLES TO relatorio_user;

-- Revogar privilégios em nível de schema
REVOKE ALL PRIVILEGES ON SCHEMA administracao FROM relatorio_user;
REVOKE ALL PRIVILEGES ON ALL TABLES IN SCHEMA administracao FROM
relatorio_user;
```

Exercício 4: Criação e Gerenciamento de Roles

Descrição

Pratique a criação e gerenciamento básico de roles.

Requisitos

- Criar roles para diferentes funções
- Conceder privilégios às roles
- Atribuir roles a usuários
- Revogar roles de usuários

Solução

```
-- Criar roles para diferentes funções
CREATE ROLE vendas_role;
CREATE ROLE estoque_role;
CREATE ROLE relatorios_role;

-- Conceder privilégios às roles
GRANT SELECT, INSERT, UPDATE ON aplicacao.pedidos TO vendas_role;
GRANT SELECT, UPDATE ON aplicacao.produtos TO estoque_role;
GRANT SELECT ON ALL TABLES IN SCHEMA relatorios TO
relatorios_role;

-- Atribuir roles a usuários
GRANT vendas_role TO app_user;
GRANT estoque_role TO app_user;
GRANT relatorios_role TO relatorio_user;

-- Revogar roles de usuários
REVOKE estoque_role FROM app_user;
```

Exercício 5: Privilégios em Nível de Coluna

Descrição

Pratique a concessão de privilégios em nível de coluna.

Requisitos

- Conceder SELECT em colunas específicas
- Conceder UPDATE em colunas específicas
- Revogar privilégios em nível de coluna
- Verificar privilégios em nível de coluna

Solução

```
-- Conceder SELECT em colunas específicas
GRANT SELECT (nome, email) ON aplicacao.clientes TO app_user;

-- Conceder UPDATE em colunas específicas
GRANT UPDATE (estoque) ON aplicacao.produtos TO app_user;

-- Revogar privilégios em nível de coluna
REVOKE UPDATE (preco) ON aplicacao.produtos FROM app_user;

-- Verificar privilégios em nível de coluna
SELECT grantee, table_schema, table_name, column_name,
privilege_type
FROM information_schema.column_privileges
WHERE grantee = 'app_user';
```

Exercício 6: Privilégios de Execução

Descrição

Pratique a concessão de privilégios para execução de funções e procedimentos.

Requisitos

- Criar função/procedimento
- Conceder privilégio de execução
- Revogar privilégio de execução
- Verificar privilégios de execução

Solução

```
-- Criar função
CREATE OR REPLACE FUNCTION
aplicacao.calcular_total_pedido(pedido_id INT)
RETURNS DECIMAL AS $$
DECLARE
```

```

total DECIMAL(10,2);
BEGIN
    SELECT SUM(quantidade * preco_unitario) INTO total
    FROM aplicacao.itens_pedido
    WHERE pedido_id = pedido_id;
    RETURN total;
END;
$$ LANGUAGE plpgsql;

-- Conceder privilégio de execução
GRANT EXECUTE ON FUNCTION aplicacao.calcular_total_pedido(INT) TO
app_user;

-- Revogar privilégio de execução
REVOKE EXECUTE ON FUNCTION aplicacao.calcular_total_pedido(INT)
FROM relatorio_user;

-- Verificar privilégios de execução
SELECT grantee, routine_schema, routine_name, privilege_type
FROM information_schema.routine_privileges
WHERE grantee = 'app_user';

```

Exercício 7: Privilégios de Conexão

Descrição

Pratique a concessão de privilégios de conexão a bancos de dados.

Requisitos

- Conceder privilégio de conexão
- Revogar privilégio de conexão
- Limitar conexões por usuário
- Verificar privilégios de conexão

Solução

```
-- Conceder privilégio de conexão
GRANT CONNECT ON DATABASE segurança_db TO app_user;

-- Revogar privilégio de conexão
REVOKE CONNECT ON DATABASE segurança_db FROM dev_user;

-- Limitar conexões por usuário (PostgreSQL)
ALTER USER app_user CONNECTION LIMIT 5;

-- Verificar privilégios de conexão
SELECT * FROM pg_database d
JOIN pg_shdescription s ON d.oid = s.objoid
WHERE s.objsubid = 0 AND d.datname = 'segurança_db';
```

Exercício 8: Privilégios Temporários

Descrição

Pratique a concessão de privilégios temporários para tarefas específicas.

Requisitos

- Conceder privilégios temporários
- Implementar expiração de privilégios
- Revogar privilégios após uso
- Registrar concessões temporárias

Solução

```
-- Conceder privilégios temporários (simulação)
GRANT UPDATE ON aplicacao.produtos TO dev_user;

-- Registrar concessão temporária
INSERT INTO administracao.privilegios_temporarios
```

```

(username, privilegio, objeto, data_expiracao)
VALUES ('dev_user', 'UPDATE', 'aplicacao.produtos', CURRENT_DATE +
INTERVAL '1 day');

-- Verificar e revogar privilégios expirados
DO $$

BEGIN
    FOR r IN SELECT username, privilegio, objeto
        FROM administracao.privilegios_temporarios
        WHERE data_expiracao < CURRENT_DATE
    LOOP
        EXECUTE 'REVOKE ' || r.privilegio || ' ON ' || r.objeto ||
' FROM ' || r.username;

        -- Registrar revogação
        INSERT INTO administracao.log_acessos
        (username, tabela_acessada, tipo_operacao)
        VALUES (r.username, r.objeto, 'REVOKE_EXPIRADO');
    END LOOP;
END $$;

```

Critérios de Avaliação

- Sintaxe correta dos comandos DCL
- Aplicação do princípio do menor privilégio
- Verificação adequada após concessão/revogação
- Organização lógica de permissões
- Documentação das decisões de segurança

Exercícios Intermediários de DCL

Exercício 1: Hierarquia de Roles

Descrição

Pratique a criação de uma hierarquia de roles para organizar permissões.

Requisitos

- Criar estrutura hierárquica de roles
- Conceder permissões em diferentes níveis
- Herança de permissões
- Verificar permissões efetivas

Solução

```
-- Criar estrutura hierárquica de roles
CREATE ROLE funcionario_base;
CREATE ROLE vendedor;
CREATE ROLE gerente_vendas;
CREATE ROLE diretor_comercial;

-- Estabelecer hierarquia
GRANT funcionario_base TO vendedor;
GRANT vendedor TO gerente_vendas;
GRANT gerente_vendas TO diretor_comercial;

-- Conceder permissões em diferentes níveis
GRANT SELECT ON aplicacao.produtos TO funcionario_base;
GRANT SELECT, INSERT ON aplicacao_pedidos TO vendedor;
GRANT UPDATE ON aplicacao_clientes TO gerente_vendas;
GRANT ALL PRIVILEGES ON SCHEMA aplicacao TO diretor_comercial;

-- Atribuir roles a usuários
```

```

GRANT vendedor TO usuario_vendas;
GRANT gerente_vendas TO usuario_gerente;
GRANT diretor_comercial TO usuario_diretor;

-- Verificar permissões efetivas
SELECT r.rolname, u.username,
       has_table_privilege(u.username, 'aplicacao.produtos',
'SELECT') as pode_consultar,
       has_table_privilege(u.username, 'aplicacao_pedidos',
'INSERT') as pode_inserir,
       has_table_privilege(u.username, 'aplicacao_clientes',
'UPDATE') as pode_atualizar
FROM pg_roles r
JOIN pg_user u ON r.rolname = u.username
WHERE r.rolname IN ('usuario_vendas', 'usuario_gerente',
'usuario_diretor');

```

Exercício 2: Permissões Baseadas em Função

Descrição

Implemente um sistema de permissões baseado nas funções dos usuários na organização.

Requisitos

- Criar roles para departamentos
- Criar roles para níveis hierárquicos
- Implementar matriz de permissões
- Gerenciar usuários com múltiplas funções

Solução

```

-- Criar roles para departamentos
CREATE ROLE dept_vendas;

```

```

CREATE ROLE dept_marketing;
CREATE ROLE dept_financeiro;
CREATE ROLE dept_ti;

-- Criar roles para níveis hierárquicos
CREATE ROLE nivel_operacional;
CREATE ROLE nivel_tatico;
CREATE ROLE nivel_estrategico;

-- Implementar matriz de permissões para departamentos
-- Vendas
GRANT SELECT, INSERT ON aplicacao.clientes TO dept_vendas;
GRANT SELECT, INSERT ON aplicacao_pedidos TO dept_vendas;
GRANT SELECT ON aplicacao_produtos TO dept_vendas;

-- Marketing
GRANT SELECT ON aplicacao.clientes TO dept_marketing;
GRANT SELECT ON aplicacao_pedidos TO dept_marketing;
GRANT SELECT ON relatorios.vendas_mensais TO dept_marketing;

-- Financeiro
GRANT SELECT ON aplicacao_pedidos TO dept_financeiro;
GRANT SELECT ON aplicacao.clientes TO dept_financeiro;
GRANT SELECT, UPDATE ON relatorios.vendas_mensais TO
dept_financeiro;

-- TI
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA aplicacao TO dept_ti;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA relatorios TO
dept_ti;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA administracao TO
dept_ti;

-- Implementar matriz de permissões para níveis
-- Operacional
GRANT SELECT ON aplicacao_produtos TO nivel_operacional;
GRANT SELECT ON aplicacao.clientes TO nivel_operacional;

```

```

-- Tático
GRANT SELECT, INSERT, UPDATE ON aplicacao.produtos TO
nivel_tatico;
GRANT SELECT, INSERT, UPDATE ON aplicacao.clientes TO
nivel_tatico;
GRANT SELECT ON relatorios.vendas_mensais TO nivel_tatico;

-- Estratégico
GRANT ALL PRIVILEGES ON SCHEMA relatorios TO nivel_estrategico;
GRANT SELECT ON ALL TABLES IN SCHEMA aplicacao TO
nivel_estrategico;

-- Gerenciar usuários com múltiplas funções
CREATE USER maria WITH PASSWORD 'senha123';
CREATE USER joao WITH PASSWORD 'senha123';
CREATE USER carlos WITH PASSWORD 'senha123';

-- Maria: Vendas + Operacional
GRANT dept_vendas TO maria;
GRANT nivel_operacional TO maria;

-- João: Marketing + Tático
GRANT dept_marketing TO joao;
GRANT nivel_tatico TO joao;

-- Carlos: TI + Estratégico
GRANT dept_ti TO carlos;
GRANT nivel_estrategico TO carlos;

```

Exercício 3: Permissões com Condições

Descrição

Implemente permissões com condições específicas usando roles e esquemas.

Requisitos

- Criar roles com permissões condicionais
- Implementar permissões por horário de acesso
- Implementar permissões por tipo de operação
- Verificar efetividade das permissões

Solução

```
-- Criar roles para diferentes condições
CREATE ROLE acesso_horario_comercial;
CREATE ROLE acesso_somente_leitura;
CREATE ROLE acesso_manutencao;

-- Conceder permissões básicas
GRANT CONNECT ON DATABASE segurança_db TO
acesso_horario_comercial;
GRANT USAGE ON SCHEMA aplicacao TO acesso_horario_comercial;
GRANT SELECT, INSERT, UPDATE ON ALL TABLES IN SCHEMA aplicacao TO
acesso_horario_comercial;

GRANT CONNECT ON DATABASE segurança_db TO acesso_somente_leitura;
GRANT USAGE ON SCHEMA aplicacao, relatorios TO
acesso_somente_leitura;
GRANT SELECT ON ALL TABLES IN SCHEMA aplicacao, relatorios TO
acesso_somente_leitura;

GRANT CONNECT ON DATABASE segurança_db TO acesso_manutencao;
GRANT USAGE ON SCHEMA aplicacao TO acesso_manutencao;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA aplicacao TO
acesso_manutencao;

-- Criar usuários para teste
CREATE USER usuario_dia WITH PASSWORD 'senha123';
CREATE USER usuario_consulta WITH PASSWORD 'senha123';
CREATE USER usuario_manutencao WITH PASSWORD 'senha123';

-- Atribuir roles aos usuários
```

```

GRANT acesso_horario_comercial TO usuario_dia;
GRANT acesso_somente_leitura TO usuario_consulta;
GRANT acesso_manutencao TO usuario_manutencao;

-- Criar tabela para controle de acesso
CREATE TABLE administracao.controle_acesso (
    id SERIAL PRIMARY KEY,
    role_name VARCHAR(50),
    horario_inicio TIME,
    horario_fim TIME,
    dias_semana VARCHAR(20),
    ativo BOOLEAN DEFAULT TRUE
);

-- Inserir regras de controle
INSERT INTO administracao.controle_acesso
(role_name, horario_inicio, horario_fim, dias_semana)
VALUES
('acesso_horario_comercial', '08:00:00', '18:00:00', 'Mon-Fri'),
('acesso_manutencao', '22:00:00', '06:00:00', 'Mon-Sun');

-- Criar função para verificar permissão de acesso
CREATE OR REPLACE FUNCTION
administracao.verificar_acesso_permitido(p_role VARCHAR)
RETURNS BOOLEAN AS $$

DECLARE
    v_permitido BOOLEAN := FALSE;
    v_regra RECORD;
    v_hora_atual TIME := CURRENT_TIME;
    v_dia_atual VARCHAR := to_char(CURRENT_DATE, 'Dy');

BEGIN
    -- Verificar se existe regra para a role
    SELECT * INTO v_regra
    FROM administracao.controle_acesso
    WHERE role_name = p_role AND ativo = TRUE;

    IF NOT FOUND THEN
        -- Se não há regra específica, acesso é permitido

```

```

        RETURN TRUE;
    END IF;

    -- Verificar horário
    IF (v_hora_atual BETWEEN v_regra.horario_inicio AND
v_regra.horario_fim) THEN
        -- Verificar dia da semana
        IF v_regra.dias_semana = 'Mon-Sun' OR
            position(v_dia_atual in v_regra.dias_semana) > 0 THEN
                v_permitido := TRUE;
            END IF;
    END IF;

    -- Registrar tentativa de acesso
    INSERT INTO administracao.log_acessos
    (username, tabela_acessada, tipo_operacao)
    VALUES (
        current_user,
        'Verificação de acesso',
        CASE WHEN v_permitido THEN 'ACESSO_PERMITIDO' ELSE
        'ACESSO_NEGADO' END
    );
    RETURN v_permitido;
END;
$$ LANGUAGE plpgsql;

-- Instruções para uso (comentadas)
/*
-- Como administrador, criar regra para verificar acesso antes de
operações
CREATE OR REPLACE RULE check_acesso_clientes AS
ON SELECT TO aplicacao.clientes
WHERE NOT administracao.verificar_acesso_permitido(current_user)
DO INSTEAD NOTHING;

-- Testar acesso em diferentes horários
-- (executar como diferentes usuários)

```

```
SELECT * FROM aplicacao.clientes;  
*/
```

Exercício 4: Auditoria de Permissões

Descrição

Implemente um sistema para auditar e monitorar permissões e acessos.

Requisitos

- Criar tabelas de auditoria
- Registrar alterações de permissões
- Criar relatórios de auditoria
- Detectar anomalias de permissões

Solução

```
-- Criar tabelas de auditoria  
CREATE TABLE administracao.auditoria_permissoes (  
    id SERIAL PRIMARY KEY,  
    data_hora TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
    usuario_executor VARCHAR(50),  
    comando TEXT,  
    objeto_afetado TEXT,  
    usuario_afetado VARCHAR(50),  
    tipo_operacao VARCHAR(20)  
);  
  
-- Criar função para registrar alterações de permissões  
manualmente  
CREATE OR REPLACE FUNCTION  
administracao.registrar_alteracao_permissao(  
    p_comando TEXT,  
    p_objeto TEXT,
```

```

    p_usuario TEXT,
    p_tipo VARCHAR
) RETURNS void AS $$

BEGIN
    INSERT INTO administracao.auditoria_permissoes
    (usuario_executor, comando, objeto_afetado, usuario_afetado,
    tipo_operacao)
    VALUES (
        current_user,
        p_comando,
        p_objeto,
        p_usuario,
        p_tipo
    );
END;
$$ LANGUAGE plpgsql;

-- Criar relatório de auditoria
CREATE OR REPLACE VIEW administracao.relatorio_permissoes AS
SELECT
    u.username AS usuario,
    n.nspname AS schema,
    c.relname AS tabela,
    c.relkind AS tipo_objeto,
    string_agg(p.privilege_type, ', ') AS privilegios
FROM pg_user u
CROSS JOIN pg_class c
JOIN pg_namespace n ON c.relnamespace = n.oid
JOIN information_schema.table_privileges p ON
    p.grantee = u.username AND
    p.table_schema = n.nspname AND
    p.table_name = c.relname
WHERE n.nspname NOT LIKE 'pg_%' AND n.nspname != 'information_schema'
GROUP BY u.username, n.nspname, c.relname, c.relkind
ORDER BY u.username, n.nspname, c.relname;

-- Detectar anomalias de permissões

```

```

CREATE OR REPLACE VIEW administracao.anomalias_permissoes AS
-- Usuários com permissões diretas (não via roles)
SELECT
    p.grantee AS usuario,
    p.table_schema || '.' || p.table_name AS objeto,
    p.privilege_type AS privilegio,
    'Permissão direta' AS tipo_anomalia
FROM information_schema.table_privileges p
WHERE p.grantee NOT IN (
    SELECT rolname FROM pg_roles WHERE rolcanlogin = false
)
AND p.table_schema NOT LIKE 'pg_%'
AND p.table_schema != 'information_schema'

UNION ALL

-- Usuários com permissões excessivas
SELECT
    p.grantee AS usuario,
    p.table_schema || '.' || p.table_name AS objeto,
    string_agg(p.privilege_type, ', ') AS privilegio,
    'Permissões excessivas' AS tipo_anomalia
FROM information_schema.table_privileges p
WHERE p.privilege_type IN ('DELETE', 'TRUNCATE')
AND p.table_schema = 'aplicacao'
GROUP BY p.grantee, p.table_schema, p.table_name;

```

Exercício 5: Gerenciamento de Permissões em Massa

Descrição

Implemente scripts para gerenciar permissões em massa para múltiplos objetos e usuários.

Requisitos

- Conceder permissões em massa

- Revogar permissões em massa
- Sincronizar permissões entre ambientes
- Documentar permissões concedidas

Solução

```
-- Função para conceder permissões em massa
CREATE OR REPLACE FUNCTION
administracao.conceder_permissoes_em_massa(
    p_schema TEXT,
    p_role TEXT,
    p_privilegios TEXT
) RETURNS void AS $$

DECLARE
    v_tabela TEXT;
BEGIN
    FOR v_tabela IN
        SELECT table_name
        FROM information_schema.tables
        WHERE table_schema = p_schema
        AND table_type = 'BASE TABLE'
    LOOP
        EXECUTE 'GRANT ' || p_privilegios || ' ON ' ||
            p_schema || '.' || v_tabela ||
            ' TO ' || p_role;

        -- Registrar na auditoria
        INSERT INTO administracao.auditoria_permissoes
        (usuario_executor, comando, objeto_afetado,
        usuario_afetado, tipo_operacao)
        VALUES (
            current_user,
            'GRANT ' || p_privilegios || ' ON ' || p_schema || '.'
            || v_tabela || ' TO ' || p_role,
            p_schema || '.' || v_tabela,
            p_role,
```

```

        'GRANT_MASSA'
    );
END LOOP;
END;
$$ LANGUAGE plpgsql;

-- Função para revogar permissões em massa
CREATE OR REPLACE FUNCTION
administracao.revogar_permissoes_em_massa(
    p_schema TEXT,
    p_role TEXT,
    p_privilegios TEXT DEFAULT 'ALL PRIVILEGES'
) RETURNS void AS $$

DECLARE
    v_tabela TEXT;
BEGIN
    FOR v_tabela IN
        SELECT table_name
        FROM information_schema.tables
        WHERE table_schema = p_schema
        AND table_type = 'BASE TABLE'
    LOOP
        EXECUTE 'REVOKE ' || p_privilegios || ' ON ' ||
            p_schema || '.' || v_tabela ||
            ' FROM ' || p_role;

        -- Registrar na auditoria
        INSERT INTO administracao.auditoria_permissoes
        (usuario_executor, comando, objeto_afetado,
        usuario_afetado, tipo_operacao)
        VALUES (
            current_user,
            'REVOKE ' || p_privilegios || ' ON ' || p_schema || |
            '.' || v_tabela || ' FROM ' || p_role,
            p_schema || '.' || v_tabela,
            p_role,
            'REVOKE_MASSA'
        );
    END LOOP;
END;
$$ LANGUAGE plpgsql;

```

```

    END LOOP;
END;
$$ LANGUAGE plpgsql;

-- Função para exportar permissões (para sincronização entre
ambientes)
CREATE OR REPLACE FUNCTION administracao.exportar_permissoes(
    p_schema TEXT DEFAULT NULL
) RETURNS TABLE (comando TEXT) AS $$

BEGIN
    RETURN QUERY
    WITH perms AS (
        SELECT
            'GRANT ' || string_agg(privilege_type, ', ') || ''
            ' ON ' || table_schema || '.' || table_name ||
            ' TO ' || grantee || ';' AS cmd
        FROM information_schema.table_privileges
        WHERE (p_schema IS NULL OR table_schema = p_schema)
        AND table_schema NOT LIKE 'pg_%'
        AND table_schema != 'information_schema'
        GROUP BY table_schema, table_name, grantee

        UNION ALL

        SELECT
            'GRANT ' || string_agg(privilege_type, ', ') || ''
            ' ON SCHEMA ' || table_schema ||
            ' TO ' || grantee || ';' AS cmd
        FROM information_schema.schema_privileges
        WHERE (p_schema IS NULL OR table_schema = p_schema)
        AND table_schema NOT LIKE 'pg_%'
        AND table_schema != 'information_schema'
        GROUP BY table_schema, grantee
    )
    SELECT cmd FROM perms
    ORDER BY cmd;

```

```
END;  
$$ LANGUAGE plpgsql;
```

Exercícios Avançados de DCL

Exercício 1: Implementação de Políticas de Segurança

Descrição

Implemente políticas de segurança abrangentes para um sistema empresarial.

Requisitos

- Definir políticas por departamento
- Implementar controle de acesso granular
- Estabelecer hierarquia de permissões
- Documentar políticas implementadas

Solução

```
-- Criar roles para departamentos
CREATE ROLE dept_vendas_role;
CREATE ROLE dept_financeiro_role;
CREATE ROLE dept_rh_role;
CREATE ROLE dept_ti_role;

-- Criar roles para níveis hierárquicos
CREATE ROLE nivel_operacional_role;
CREATE ROLE nivel_gerencial_role;
CREATE ROLE nivel_diretoria_role;

-- Estabelecer hierarquia
GRANT nivel_operacional_role TO nivel_gerencial_role;
GRANT nivel_gerencial_role TO nivel_diretoria_role;

-- Definir permissões por departamento - Vendas
GRANT USAGE ON SCHEMA aplicacao TO dept_vendas_role;
GRANT SELECT, INSERT ON aplicacao.clientes TO dept_vendas_role;
```

```
GRANT SELECT, INSERT ON aplicacao.pedidos TO dept_vendas_role;
GRANT SELECT ON aplicacao.produtos TO dept_vendas_role;

-- Definir permissões por departamento - Financeiro
GRANT USAGE ON SCHEMA aplicacao, relatorios TO
dept_financeiro_role;
GRANT SELECT ON aplicacao.pedidos TO dept
```

Laboratórios de DCL

Visão Geral

Os laboratórios práticos de DCL (Data Control Language) são projetados para fornecer experiência hands-on com operações de controle de acesso em ambientes PostgreSQL. Cada laboratório apresenta cenários reais e desafios práticos comumente encontrados em ambientes de produção.

Estrutura dos Laboratórios

Cada laboratório segue uma estrutura consistente:

1. Preparação do Ambiente

- Configuração inicial
- Pré-requisitos
- Scripts de setup

2. Objetivos de Aprendizado

- Conceitos principais
- Habilidades técnicas
- Resultados esperados

3. Roteiro Prático

- Instruções passo a passo
- Comandos e scripts
- Pontos de verificação

4. Avaliação

- Critérios de conclusão

- Testes de validação
- Métricas de sucesso

Ambiente de Laboratório

```
-- Criar database dedicado para laboratórios
CREATE DATABASE lab_dcl;

-- Schema para isolamento de exercícios
CREATE SCHEMA lab_workspace;

-- Tabelas para os laboratórios
CREATE TABLE lab_workspace.clientes (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    data_cadastro DATE DEFAULT CURRENT_DATE,
    status VARCHAR(20) DEFAULT 'Ativo',
    segmento VARCHAR(50),
    limite_credito DECIMAL(10, 2) DEFAULT 1000.00,
    dados_sensíveis TEXT
);

CREATE TABLE lab_workspace.produtos (
    id SERIAL PRIMARY KEY,
    codigo VARCHAR(20) UNIQUE,
    nome VARCHAR(100) NOT NULL,
    descricao TEXT,
    preco DECIMAL(10, 2) NOT NULL CHECK (preco > 0),
    estoque INTEGER DEFAULT 0,
    categoria VARCHAR(50),
    data_cadastro DATE DEFAULT CURRENT_DATE,
    status VARCHAR(20) DEFAULT 'Ativo'
);

CREATE TABLE lab_workspace.pedidos (
```

```

        id SERIAL PRIMARY KEY,
        cliente_id INTEGER REFERENCES lab_workspace.clientes(id),
        data_pedido TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        valor_total DECIMAL(10, 2) DEFAULT 0,
        status VARCHAR(20) DEFAULT 'Pendente',
        observacoes TEXT
);

CREATE TABLE lab_workspace.itens_pedido (
    id SERIAL PRIMARY KEY,
    pedido_id INTEGER REFERENCES lab_workspace.pedidos(id),
    produto_id INTEGER REFERENCES lab_workspace.produtos(id),
    quantidade INTEGER NOT NULL CHECK (quantidade > 0),
    preco_unitario DECIMAL(10, 2) NOT NULL,
    desconto DECIMAL(5, 2) DEFAULT 0
);

CREATE TABLE lab_workspace.usuarios (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    nome_completo VARCHAR(100) NOT NULL,
    departamento VARCHAR(50),
    cargo VARCHAR(50),
    data_criacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    ultimo_acesso TIMESTAMP,
    status VARCHAR(20) DEFAULT 'Ativo'
);

CREATE TABLE lab_workspace.log_acessos (
    id SERIAL PRIMARY KEY,
    usuario_id INTEGER REFERENCES lab_workspace.usuarios(id),
    data_hora TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    operacao VARCHAR(20),
    tabela_acessada VARCHAR(50),
    detalhes TEXT
);

-- Tabela de controle de progresso

```

```
CREATE TABLE lab_workspace.lab_progress (
    lab_id SERIAL PRIMARY KEY,
    lab_name VARCHAR(100),
    start_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    completion_time TIMESTAMP,
    status VARCHAR(20) DEFAULT 'IN_PROGRESS',
    notes TEXT
);
```

Laboratórios Disponíveis

1. Gerenciamento de Usuários

- Criação e configuração de usuários
- Atribuição de permissões básicas
- Gerenciamento de senhas e políticas
- Monitoramento de atividades

2. Hierarquia de Roles

- Implementação de estrutura de roles
- Herança de permissões
- Separação de responsabilidades
- Melhores práticas de design

3. Auditoria de Segurança

- Configuração de logs de auditoria
- Monitoramento de atividades suspeitas
- Análise de padrões de acesso

- Relatórios de conformidade

4. Migração de Permissões

- Exportação de permissões existentes
- Planejamento de migração
- Implementação de novas políticas
- Validação e testes de acesso

Fluxo de Trabalho Recomendado

Boas Práticas

DIRETRIZES DE SEGURANÇA:

- ✓ Privilégio mínimo necessário
- ✓ Separação de responsabilidades
- ✓ Auditoria regular
- ✓ Documentação de políticas
- ✓ Testes de penetração

Ferramentas Necessárias

1. PostgreSQL Client

- psql
- pgAdmin 4
- DBeaver

2. Utilitários de Segurança

- pg_dump (para backup de permissões)

- pgAudit (para auditoria avançada)
- Ferramentas de análise de logs

3. Scripts de Suporte

```
-- Script de verificação de ambiente
CREATE OR REPLACE FUNCTION
lab_workspace.check_security_environment()
RETURNS TABLE (
    check_item VARCHAR,
    status VARCHAR,
    details TEXT
) AS $$

BEGIN
    RETURN QUERY
    SELECT 'Database Version'::VARCHAR,
           version()::VARCHAR,
           'Verificação da versão do PostgreSQL'::TEXT
    UNION ALL
    SELECT 'Authentication Method',
           current_setting('password_encryption'),
           'Método de criptografia de senhas'
    UNION ALL
    SELECT 'SSL Status',
           CASE WHEN current_setting('ssl') = 'on' THEN
               'Enabled'
           ELSE
               'Disabled'
           END,
           'Status da conexão SSL';
END;
$$ LANGUAGE plpgsql;
```

Navegação dos Laboratórios

- Laboratório de Gerenciamento de Usuários ([Laboratório: Gerenciamento de Usuários](#))
- Laboratório de Hierarquia de Roles ([Laboratório: Hierarquia de Roles](#))
- Laboratório de Auditoria de Segurança ([Laboratório: Auditoria de Segurança](#))

- Laboratório de Migração de Permissões ([Laboratório: Migração de Permissões](#))

Supporte e Recursos

Documentação

- PostgreSQL Official Documentation (<https://www.postgresql.org/docs/>)
- DCL Reference Guide (<https://www.postgresql.org/docs/current/ddl-priv.html>)
- Security Best Practices (<https://www.postgresql.org/docs/current/user-manag.html>)

Comunidade

- Fórum PostgreSQL
- Stack Overflow
- GitHub Discussions

Conclusão

Os laboratórios DCL fornecem uma base prática essencial para o desenvolvimento de habilidades em segurança de banco de dados. A prática regular destes exercícios contribuirá significativamente para sua expertise em controle de acesso e segurança em ambientes PostgreSQL.

Próximos Passos

1. Preparação

- Configure seu ambiente local
- Revise os pré-requisitos
- Familiarize-se com as ferramentas

2. Execução

- Siga os laboratórios em ordem
- Complete todos os exercícios
- Documente seus resultados

3. Avançado

- Explore variações dos exercícios
- Crie seus próprios cenários
- Compartilhe experiências



Nota: Certifique-se de manter backups e usar ambientes de teste apropriados durante a execução dos laboratórios. Nunca pratique operações de segurança em ambientes de produção sem a devida autorização e planejamento.

Laboratório: Gerenciamento de Usuários

Objetivo

Praticar a criação, configuração e gerenciamento de usuários em um banco de dados PostgreSQL, aplicando boas práticas de segurança e controle de acesso.

Cenário

Você é um administrador de banco de dados em uma empresa que está implementando um novo sistema de gerenciamento de permissões. Sua tarefa é configurar usuários com diferentes níveis de acesso de acordo com suas funções na organização.

Setup Inicial

```
-- Conectar ao banco de dados como superusuário
-- psql -U postgres -d lab_dcl

-- Verificar usuários existentes
SELECT username, usecreaterdb, usesuper, passwd IS NOT NULL AS
has_password
FROM pg_catalog.pg_user;

-- Registrar início do laboratório
INSERT INTO lab_workspace.lab_progress (lab_name)
VALUES ('Gerenciamento de Usuários');
```

Parte 1: Criação de Usuários Básicos

1.1 Criação de Usuários

```
-- Criar usuário básico
CREATE USER app_user WITH PASSWORD 'app123';
```

```
-- Criar usuário com data de expiração
CREATE USER temp_user WITH
    PASSWORD 'temp456'
    VALID UNTIL '2023-12-31 23:59:59';

-- Criar usuário sem privilégios de login
CREATE USER app_service WITH
    PASSWORD 'srv789'
    NOLOGIN;

-- Verificar usuários criados
SELECT username, usesuper, usecreatedb, valuntil
FROM pg_catalog.pg_user
WHERE username IN ('app_user', 'temp_user', 'app_service');
```

1.2 Configuração de Parâmetros

```
-- Alterar configurações de usuário
ALTER USER app_user SET search_path TO lab_workspace, public;
ALTER USER app_user SET statement_timeout = '30s';
ALTER USER app_user SET idle_in_transaction_session_timeout = '60s';

-- Verificar configurações
SELECT username, useconfig
FROM pg_catalog.pg_user
WHERE username = 'app_user';
```

Parte 2: Atribuição de Permissões

2.1 Permissões Básicas

```
-- Conceder permissões de conexão
GRANT CONNECT ON DATABASE lab_dcl TO app_user;
GRANT CONNECT ON DATABASE lab_dcl TO temp_user;
```

```
-- Conceder permissões de uso no schema  
GRANT USAGE ON SCHEMA lab_workspace TO app_user;  
GRANT USAGE ON SCHEMA lab_workspace TO temp_user;  
  
-- Conceder permissões de leitura para app_user  
GRANT SELECT ON ALL TABLES IN SCHEMA lab_workspace TO app_user;  
  
-- Conceder permissões específicas para temp_user  
GRANT SELECT ON lab_workspace.clientes TO temp_user;  
GRANT SELECT ON lab_workspace.produtos TO temp_user;
```

2.2 Permissões Avançadas

```
-- Conceder permissões de modificação em tabelas específicas  
GRANT INSERT, UPDATE ON lab_workspace.clientes TO app_user;  
GRANT INSERT, UPDATE, DELETE ON lab_workspace_pedidos TO app_user;  
GRANT INSERT, UPDATE, DELETE ON lab_workspace_itens_pedido TO  
app_user;  
  
-- Conceder permissões em nível de coluna  
GRANT UPDATE (nome, email, status) ON lab_workspace.clientes TO  
app_user;  
GRANT UPDATE (status, observacoes) ON lab_workspace_pedidos TO  
temp_user;  
  
-- Revogar permissões sensíveis  
REVOKE UPDATE (limite_credito, dados_sensíveis) ON  
lab_workspace.clientes FROM app_user;
```

Parte 3: Gerenciamento de Senhas e Políticas

3.1 Alteração de Senhas

```
-- Alterar senha de usuário  
ALTER USER app_user WITH PASSWORD 'novasenha123';
```

```
-- Forçar expiração de senha
ALTER USER temp_user VALID UNTIL '2023-10-31 23:59:59';

-- Desabilitar usuário temporariamente
ALTER USER temp_user WITH NOLOGIN;
```

3.2 Implementação de Políticas

```
-- Criar função para registrar acessos
CREATE OR REPLACE FUNCTION lab_workspace.log_user_access()
RETURNS trigger AS $$

BEGIN
    INSERT INTO lab_workspace.log_acessos (
        usuario_id,
        operacao,
        tabela_acessada,
        detalhes
    ) VALUES (
        (SELECT id FROM lab_workspace.usuarios WHERE username = current_user),
        TG_OP,
        TG_TABLE_NAME,
        'Acesso em ' || now()
    );
    RETURN NEW;
END;

$$ LANGUAGE plpgsql;

-- Criar trigger para auditoria
CREATE TRIGGER log_clientes_access
AFTER INSERT OR UPDATE OR DELETE ON lab_workspace.clientes
FOR EACH STATEMENT EXECUTE FUNCTION
lab_workspace.log_user_access();
```

Parte 4: Monitoramento e Auditoria

4.1 Visualização de Atividades

```
-- Consultar atividades de usuários
SELECT username, application_name, client_addr, backend_start,
state
FROM pg_stat_activity
WHERE username IN ('app_user', 'temp_user');

-- Consultar permissões concedidas
SELECT grantee, table_schema, table_name, privilege_type
FROM information_schema.table_privileges
WHERE grantee IN ('app_user', 'temp_user')
ORDER BY grantee, table_schema, table_name;
```

4.2 Relatórios de Segurança

```
-- Criar visão para relatório de permissões
CREATE OR REPLACE VIEW lab_workspace.user_permissions_report AS
SELECT
    u.username AS username,
    s.nspname AS schema,
    t.relname AS table,
    array_agg(p.privilege_type) AS privileges
FROM pg_user u
CROSS JOIN pg_namespace s
JOIN pg_class t ON t.relnamespace = s.oid
JOIN information_schema.table_privileges p ON
    p.table_schema = s.nspname AND
    p.table_name = t.relname AND
    p.grantee = u.username
WHERE s.nspname = 'lab_workspace'
GROUP BY u.username, s.nspname, t.relname
ORDER BY u.username, s.nspname, t.relname;

-- Consultar relatório
SELECT * FROM lab_workspace.user_permissions_report;
```

Verificações e Testes

1. Conecte-se como cada usuário e verifique suas permissões:

```
psql -U app_user -d lab_dcl
```

2. Tente executar operações permitidas e não permitidas:

```
-- Como app_user
SELECT * FROM lab_workspace.clientes;
UPDATE lab_workspace.clientes SET nome = 'Teste' WHERE id = 1;
UPDATE lab_workspace.clientes SET limite_credito = 5000 WHERE id
= 1; -- Deve falhar
```

3. Verifique os logs de auditoria:

```
SELECT * FROM lab_workspace.log_acessos ORDER BY data_hora DESC;
```

Conclusão

Neste laboratório, você praticou:

- Criação e configuração de usuários
- Atribuição de permissões em diferentes níveis
- Gerenciamento de senhas e políticas de segurança
- Monitoramento e auditoria de atividades

Estas habilidades são fundamentais para implementar um sistema de controle de acesso robusto e seguro em ambientes PostgreSQL.

Próximos Passos

1. Explore a criação de roles para agrupar permissões

2. Implemente políticas de segurança mais avançadas
3. Configure autenticação externa (LDAP, SSO)
4. Desenvolva um sistema de auditoria mais abrangente

⚠ Nota: Registre a conclusão do laboratório:

```
UPDATE lab_workspace.lab_progress
SET completion_time = CURRENT_TIMESTAMP, status =
'COMPLETED'
WHERE lab_name = 'Gerenciamento de Usuários';
```

Laboratório: Hierarquia de Roles

Objetivo

Implementar uma estrutura hierárquica de roles em um banco de dados PostgreSQL, aplicando o princípio de separação de responsabilidades e herança de permissões.

Cenário

Você é um arquiteto de segurança em uma empresa com múltiplos departamentos e níveis hierárquicos. Sua tarefa é implementar uma estrutura de roles que reflita a organização da empresa e facilite o gerenciamento de permissões.

Setup Inicial

```
-- Conectar ao banco de dados como superusuário
-- psql -U postgres -d lab_dcl

-- Verificar roles existentes
SELECT rolname, rolsuper, rolinherit, rolcanlogin
FROM pg_roles
WHERE rolname NOT LIKE 'pg_%';

-- Registrar início do laboratório
INSERT INTO lab_workspace.lab_progress (lab_name)
VALUES ('Hierarquia de Roles');
```

Parte 1: Criação de Estrutura Básica de Roles

1.1 Roles Departamentais

```
-- Criar roles para departamentos
CREATE ROLE dept_vendas;
CREATE ROLE dept_marketing;
CREATE ROLE dept_financeiro;
CREATE ROLE dept_ti;
```

```
-- Verificar roles criadas
SELECT rolname FROM pg_roles WHERE rolname LIKE 'dept_%';
```

1.2 Roles Hierárquicas

```
-- Criar roles para níveis hierárquicos
CREATE ROLE nivel_operacional;
CREATE ROLE nivel_tatico;
CREATE ROLE nivel_estrategico;

-- Criar roles para funções específicas
CREATE ROLE analista;
CREATE ROLE supervisor;
CREATE ROLE gerente;
CREATE ROLE diretor;

-- Estabelecer hierarquia
GRANT analista TO nivel_operacional;
GRANT supervisor TO nivel_tatico;
GRANT gerente TO nivel_tatico;
GRANT diretor TO nivel_estrategico;
```

Parte 2: Implementação de Herança de Permissões

2.1 Permissões Departamentais

```
-- Conceder permissões aos departamentos
-- Vendas
GRANT USAGE ON SCHEMA lab_workspace TO dept_vendas;
GRANT SELECT, INSERT ON lab_workspace.clientes TO dept_vendas;
GRANT SELECT, INSERT, UPDATE ON lab_workspace_pedidos TO
dept_vendas;
GRANT SELECT, INSERT, UPDATE ON lab_workspace_itens_pedido TO
dept_vendas;
GRANT SELECT ON lab_workspace_produtos TO dept_vendas;
```

```

-- Marketing
GRANT USAGE ON SCHEMA lab_workspace TO dept_marketing;
GRANT SELECT ON lab_workspace.clientes TO dept_marketing;
GRANT SELECT ON lab_workspace.pedidos TO dept_marketing;
GRANT SELECT ON lab_workspace.produtos TO dept_marketing;

-- Financeiro
GRANT USAGE ON SCHEMA lab_workspace TO dept_financeiro;
GRANT SELECT ON lab_workspace.clientes TO dept_financeiro;
GRANT SELECT ON lab_workspace.pedidos TO dept_financeiro;
GRANT SELECT ON lab_workspace.itens_pedido TO dept_financeiro;
GRANT UPDATE (limite_credito) ON lab_workspace.clientes TO
dept_financeiro;

-- TI
GRANT USAGE ON SCHEMA lab_workspace TO dept_ti;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA lab_workspace TO
dept_ti;

```

2.2 Permissões por Nível Hierárquico

```

-- Nível Operacional
GRANT SELECT ON ALL TABLES IN SCHEMA lab_workspace TO
nivel_operacional;

-- Nível Tático
GRANT nivel_operacional TO nivel_tatico;
GRANT INSERT, UPDATE ON ALL TABLES IN SCHEMA lab_workspace TO
nivel_tatico;

-- Nível Estratégico
GRANT nivel_tatico TO nivel_estrategico;
GRANT DELETE ON ALL TABLES IN SCHEMA lab_workspace TO

```

```
nivel_estrategico;  
GRANT USAGE, CREATE ON SCHEMA lab_workspace TO nivel_estrategico;
```

Parte 3: Criação de Usuários com Múltiplas Roles

3.1 Usuários Departamentais

```
-- Criar usuários com roles departamentais e hierárquicas  
CREATE USER vendedor01 WITH PASSWORD 'v123' IN ROLE dept_vendas,  
nivel_operacional;  
CREATE USER supervisor_vendas WITH PASSWORD 's123' IN ROLE  
dept_vendas, nivel_tatico;  
CREATE USER gerente_vendas WITH PASSWORD 'g123' IN ROLE  
dept_vendas, nivel_tatico;  
CREATE USER diretor_vendas WITH PASSWORD 'd123' IN ROLE  
dept_vendas, nivel_estrategico;  
  
CREATE USER analista_marketing WITH PASSWORD 'a123' IN ROLE  
dept_marketing, nivel_operacional;  
CREATE USER gerente_marketing WITH PASSWORD 'g123' IN ROLE  
dept_marketing, nivel_tatico;  
  
CREATE USER analista_financeiro WITH PASSWORD 'a123' IN ROLE  
dept_financeiro, nivel_operacional;  
CREATE USER controller WITH PASSWORD 'c123' IN ROLE  
dept_financeiro, nivel_tatico;  
  
CREATE USER admin_ti WITH PASSWORD 'a123' IN ROLE dept_ti,  
nivel_estrategico;
```

3.2 Verificação de Membros de Roles

```
-- Verificar membros de cada role  
SELECT r.rolname, u.username  
FROM pg_roles r  
JOIN pg_auth_members m ON r.oid = m.roleid
```

```
JOIN pg_roles u ON m.member = u.oid
WHERE r.rolname LIKE 'dept_%' OR r.rolname LIKE 'nivel_%
ORDER BY r.rolname, u.username;
```

Parte 4: Implementação de Roles Funcionais

4.1 Roles para Funções Específicas

```
-- Criar roles para funções específicas
CREATE ROLE atendimento_cliente;
GRANT SELECT, INSERT, UPDATE ON lab_workspace.clientes TO
atendimento_cliente;
GRANT SELECT ON lab_workspace_pedidos TO atendimento_cliente;

CREATE ROLE gestor_estoque;
GRANT SELECT, UPDATE ON lab_workspace.produtos TO gestor_estoque;

CREATE ROLE processador_pedidos;
GRANT SELECT, UPDATE ON lab_workspace_pedidos TO
processador_pedidos;
GRANT SELECT, INSERT, UPDATE ON lab_workspace_itens_pedido TO
processador_pedidos;
GRANT SELECT ON lab_workspace.produtos TO processador_pedidos;
GRANT SELECT ON lab_workspace.clientes TO processador_pedidos;

-- Atribuir roles funcionais aos usuários
GRANT atendimento_cliente TO vendedor01;
GRANT gestor_estoque TO supervisor_vendas;
GRANT processador_pedidos TO vendedor01, supervisor_vendas;
```

4.2 Roles Temporárias

```
-- Criar role para acesso temporário
CREATE ROLE acesso_auditoria;
GRANT SELECT ON ALL TABLES IN SCHEMA lab_workspace TO
acesso_auditoria;
```

```

-- Criar role com restrição de tempo
CREATE ROLE acesso_noturno;
GRANT SELECT ON ALL TABLES IN SCHEMA lab_workspace TO
acesso_noturno;

-- Implementar controle de acesso baseado em tempo
CREATE OR REPLACE FUNCTION lab_workspace.check_time_access()
RETURNS boolean AS $$

DECLARE
    current_hour integer;
BEGIN
    current_hour := EXTRACT(HOUR FROM CURRENT_TIME);
    RETURN current_hour BETWEEN 18 AND 6;
END;
$$ LANGUAGE plpgsql;

-- Nota: Em um ambiente real, você usaria Row-Level Security para
implementar isso
-- Este é apenas um exemplo conceitual

```

Parte 5: Análise e Auditoria de Roles

5.1 Visualização da Hierarquia de Roles

```

-- Criar função para visualizar hierarquia de roles
CREATE OR REPLACE FUNCTION lab_workspace.role_hierarchy(role_name
text)
RETURNS TABLE (role text, level int) AS $$

WITH RECURSIVE role_tree AS (
    SELECT r.rolname AS role, 1 AS level
    FROM pg_roles r
    WHERE r.rolname = role_name

    UNION ALL

    SELECT r.rolname, rt.level + 1
    FROM role_tree rt
    JOIN pg_roles r ON r.rolname = rt.role
)
```

```

    FROM role_tree rt
    JOIN pg_auth_members m ON rt.role = (SELECT rolname FROM
pg_roles WHERE oid = m.roleid)
    JOIN pg_roles r ON r.oid = m.member
)
SELECT role, level FROM role_tree ORDER BY level, role;
$$ LANGUAGE sql;

-- Visualizar hierarquia para um usuário específico
SELECT * FROM lab_workspace.role_hierarchy('diretor_vendas');

```

5.2 Relatório de Permissões Efetivas

```

-- Criar visão para relatório de permissões efetivas
CREATE OR REPLACE VIEW lab_workspace.effective_permissions AS
WITH role_members AS (
    SELECT r.rolname AS role, u.username AS username
    FROM pg_roles r
    JOIN pg_auth_members m ON r.oid = m.roleid
    JOIN pg_roles u ON m.member = u.oid
    WHERE u.rolcanlogin

    UNION

    SELECT r.rolname AS role, r.rolname AS username
    FROM pg_roles r
    WHERE r.rolcanlogin
)
SELECT
    rm.username,
    t.table_schema,
    t.table_name,
    t.privilege_type
FROM role_members rm
JOIN information_schema.role_table_grants t ON rm.role = t.grantee
WHERE t.table_schema = 'lab_workspace'
ORDER BY rm.username, t.table_schema, t.table_name,

```

```
t.privilege_type;

-- Consultar permissões efetivas
SELECT * FROM lab_workspace.effective_permissions WHERE username = 'vendedor01';
```

Verificações e Testes

1. Conecte-se como diferentes usuários e verifique suas permissões:

```
psql -U vendedor01 -d lab_dcl
psql -U gerente_marketing -d lab_dcl
psql -U admin_ti -d lab_dcl
```

2. Teste operações permitidas e não permitidas:

```
-- Como vendedor01
SELECT * FROM lab_workspace.clientes;
INSERT INTO lab_workspace.clientes (nome, email) VALUES ('Cliente Teste', 'teste@email.com');
DELETE FROM lab_workspace.clientes WHERE id = 1; -- Deve falhar

-- Como gerente_vendas
SELECT * FROM lab_workspace.clientes;
INSERT INTO lab_workspace.clientes (nome, email) VALUES ('Cliente Gerente', 'gerente@email.com');
UPDATE lab_workspace.clientes SET status = 'Inativo' WHERE id = 1;

-- Como admin_ti
CREATE TABLE lab_workspace.teste_admin (id serial primary key, descricao text);
DROP TABLE lab_workspace.teste_admin;
```

3. Verifique a hierarquia de roles:

```
SELECT * FROM lab_workspace.role_hierarchy('nivel_estrategico');
SELECT * FROM lab_workspace.role_hierarchy('gerente_vendas');
```

Conclusão

Neste laboratório, você implementou:

- Uma estrutura hierárquica de roles que reflete a organização da empresa
- Herança de permissões entre níveis hierárquicos
- Separação de responsabilidades por departamento
- Roles funcionais para tarefas específicas
- Ferramentas para análise e auditoria de permissões

Esta abordagem estruturada para gerenciamento de permissões oferece vários benefícios:

1. **Simplicidade de administração:** Alterações em permissões podem ser feitas em nível de role, afetando automaticamente todos os usuários associados
2. **Consistência:** Usuários com funções similares recebem permissões consistentes
3. **Segurança:** O princípio de privilégio mínimo é aplicado em cada nível
4. **Flexibilidade:** Usuários podem pertencer a múltiplas roles, combinando permissões conforme necessário

Próximos Passos

1. Implemente Row-Level Security (RLS) para controle de acesso mais granular
2. Desenvolva um sistema automatizado para gerenciamento de roles
3. Integre com sistemas de identidade externos (LDAP, Active Directory)
4. Implemente rotação automática de credenciais

⚠️ Nota: Registre a conclusão do laboratório:

```
UPDATE lab_workspace.lab_progress
SET completion_time = CURRENT_TIMESTAMP, status =
'COMPLETED'
WHERE lab_name = 'Hierarquia de Roles';
```

Laboratório: Auditoria de Segurança

Objetivo

Implementar e configurar um sistema abrangente de auditoria de segurança em um banco de dados PostgreSQL, identificando atividades suspeitas e garantindo conformidade com políticas de segurança.

Cenário

Você é um especialista em segurança de dados em uma instituição financeira que precisa implementar controles de auditoria rigorosos para atender a requisitos regulatórios e de compliance. Sua tarefa é configurar um sistema de auditoria que capture todas as atividades relevantes no banco de dados.

Setup Inicial

```
-- Conectar ao banco de dados como superusuário
-- psql -U postgres -d lab_dcl

-- Verificar extensões disponíveis
SELECT name, default_version, installed_version
FROM pg_available_extensions
WHERE name IN ('pgaudit', 'pg_stat_statements');

-- Registrar início do laboratório
INSERT INTO lab_workspace.lab_progress (lab_name)
VALUES ('Auditoria de Segurança');
```

Parte 1: Configuração de Logs Básicos

1.1 Configuração de Parâmetros de Log

```

-- Verificar configurações atuais
SHOW log_destination;
SHOW logging_collector;
SHOW log_directory;
SHOW log_filename;
SHOW log_statement;

-- Configurar parâmetros (em postgresql.conf ou via ALTER SYSTEM)
-- Nota: Em um ambiente real, você alteraria o postgresql.conf
-- Aqui usamos ALTER SYSTEM para demonstração

ALTER SYSTEM SET log_destination = 'csvlog';
ALTER SYSTEM SET logging_collector = 'on';
ALTER SYSTEM SET log_directory = 'pg_log';
ALTER SYSTEM SET log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log';
ALTER SYSTEM SET log_rotation_age = '1d';
ALTER SYSTEM SET log_rotation_size = '10MB';

-- Configurar o que será logado
ALTER SYSTEM SET log_statement = 'mod'; -- Loga todas as
modificações (INSERT, UPDATE, DELETE, etc.)
ALTER SYSTEM SET log_min_duration_statement = '1000'; -- Loga
queries que demoram mais de 1 segundo
ALTER SYSTEM SET log_connections = 'on';
ALTER SYSTEM SET log_disconnections = 'on';
ALTER SYSTEM SET log_duration = 'on';

-- Aplicar alterações (requer reinicialização do servidor)
-- SELECT pg_reload_conf();

```

1.2 Criação de Tabelas de Auditoria

```

-- Criar schema dedicado para auditoria
CREATE SCHEMA IF NOT EXISTS audit;

-- Tabela para registro de atividades
CREATE TABLE audit.activity_log (

```

```

    id SERIAL PRIMARY KEY,
    event_time TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    username TEXT NOT NULL,
    database_name TEXT NOT NULL,
    schema_name TEXT NOT NULL,
    table_name TEXT,
    action_type TEXT NOT NULL,
    query TEXT,
    old_data JSONB,
    new_data JSONB,
    client_ip TEXT,
    application_name TEXT
);

-- Tabela para registro de logins
CREATE TABLE audit.login_attempts (
    id SERIAL PRIMARY KEY,
    event_time TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    username TEXT NOT NULL,
    success BOOLEAN NOT NULL,
    client_ip TEXT,
    connection_type TEXT,
    application_name TEXT,
    details TEXT
);

-- Tabela para registro de alterações de permissões
CREATE TABLE audit.permission_changes (
    id SERIAL PRIMARY KEY,
    event_time TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    username TEXT NOT NULL,
    action_type TEXT NOT NULL,
    object_type TEXT NOT NULL,
    object_name TEXT NOT NULL,
    grantee TEXT,
    permission TEXT,

```

```
query TEXT  
);
```

Parte 2: Implementação de Triggers de Auditoria

2.1 Função Genérica de Auditoria

```
-- Criar função para auditoria de alterações em tabelas  
CREATE OR REPLACE FUNCTION audit.log_table_changes()  
RETURNS TRIGGER AS $$  
DECLARE  
    old_data JSONB := NULL;  
    new_data JSONB := NULL;  
BEGIN  
    IF TG_OP = 'DELETE' THEN  
        old_data := row_to_json(OLD)::JSONB;  
    ELSIF TG_OP = 'UPDATE' THEN  
        old_data := row_to_json(OLD)::JSONB;  
        new_data := row_to_json(NEW)::JSONB;  
    ELSIF TG_OP = 'INSERT' THEN  
        new_data := row_to_json(NEW)::JSONB;  
    END IF;  
  
    INSERT INTO audit.activity_log (  
        username,  
        database_name,  
        schema_name,  
        table_name,  
        action_type,  
        query,  
        old_data,  
        new_data,  
        client_ip,  
        application_name  
    ) VALUES (  
        current_user,  
        current_database(),
```

```

TG_TABLE_SCHEMA,
TG_TABLE_NAME,
TG_OP,
current_query(),
old_data,
new_data,
inet_client_addr(),
current_setting('application_name')
);

RETURN NULL;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;

```

2.2 Aplicação de Triggers nas Tabelas

```

-- Aplicar triggers nas tabelas principais
CREATE TRIGGER audit_clientes
AFTER INSERT OR UPDATE OR DELETE ON lab_workspace.clientes
FOR EACH ROW EXECUTE FUNCTION audit.log_table_changes();

CREATE TRIGGER audit_produtos
AFTER INSERT OR UPDATE OR DELETE ON lab_workspace.produtos
FOR EACH ROW EXECUTE FUNCTION audit.log_table_changes();

CREATE TRIGGER audit_pedidos
AFTER INSERT OR UPDATE OR DELETE ON lab_workspace.pedidos
FOR EACH ROW EXECUTE FUNCTION audit.log_table_changes();

CREATE TRIGGER audit_itens_pedido
AFTER INSERT OR UPDATE OR DELETE ON lab_workspace.itens_pedido
FOR EACH ROW EXECUTE FUNCTION audit.log_table_changes();

CREATE TRIGGER audit_usuarios

```

```
AFTER INSERT OR UPDATE OR DELETE ON lab_workspace.usuarios
FOR EACH ROW EXECUTE FUNCTION audit.log_table_changes();
```

Parte 3: Monitoramento de Atividades Sensíveis

3.1 Auditoria de Permissões

```
-- Função para auditoria de alterações de permissões
CREATE OR REPLACE FUNCTION audit.log_permission_changes()
RETURNS event_trigger AS $$

DECLARE
    obj record;
    query text;
BEGIN
    query := current_query();

    IF query ~* 'grant|revoke' THEN
        INSERT INTO audit.permission_changes (
            username,
            action_type,
            object_type,
            object_name,
            query
        ) VALUES (
            current_user,
            CASE
                WHEN query ~* 'grant' THEN 'GRANT'
                WHEN query ~* 'revoke' THEN 'REVOKE'
                ELSE 'UNKNOWN'
            END,
            CASE
                WHEN query ~* 'table' THEN 'TABLE'
                WHEN query ~* 'function' THEN 'FUNCTION'
                WHEN query ~* 'schema' THEN 'SCHEMA'
                WHEN query ~* 'sequence' THEN 'SEQUENCE'
                ELSE 'UNKNOWN'
            END,
            ...
        );
    END IF;
END;
```

```

        regexp_replace(query, '.*(?:on|ON)\s+([^\s]+).*', 
'\1'),
    query
);
END IF;
END;
$$ LANGUAGE plpgsql;

-- Criar event trigger para capturar alterações de permissões
CREATE EVENT TRIGGER permission_audit ON ddl_command_end
WHEN TAG IN ('GRANT', 'REVOKE')
EXECUTE FUNCTION audit.log_permission_changes();

```

3.2 Auditoria de Logins

```

-- Função para simular auditoria de logins
-- Nota: Em um ambiente real, isso seria implementado via
configuração do PostgreSQL
-- ou usando extensões como pgaudit

CREATE OR REPLACE FUNCTION audit.simulate_login_audit()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO audit.login_attempts (
        username,
        success,
        client_ip,
        application_name
    ) VALUES (
        NEW.username,
        TRUE,
        inet_client_addr(),
        current_setting('application_name')
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

```

-- Aplicar trigger na tabela de usuários para simular login
-- (apenas para demonstração - em ambiente real seria diferente)
CREATE TRIGGER simulate_login
AFTER UPDATE OF ultimo_acesso ON lab_workspace.usuarios
FOR EACH ROW EXECUTE FUNCTION audit.simulate_login_audit();

-- Função para registrar login manualmente (para demonstração)
CREATE OR REPLACE FUNCTION lab_workspace.register_login(username
text)
RETURNS void AS $$

BEGIN
    UPDATE lab_workspace.usuarios
    SET ultimo_acesso = CURRENT_TIMESTAMP
    WHERE username = register_login.username;
END;
$$ LANGUAGE plpgsql;

```

Parte 4: Análise e Relatórios de Auditoria

4.1 Visões para Análise

```

-- Visão para atividades por usuário
CREATE OR REPLACE VIEW audit.user_activity AS
SELECT
    username,
    COUNT(*) AS total_actions,
    COUNT(*) FILTER (WHERE action_type = 'INSERT') AS inserts,
    COUNT(*) FILTER (WHERE action_type = 'UPDATE') AS updates,
    COUNT(*) FILTER (WHERE action_type = 'DELETE') AS deletes,
    MIN(event_time) AS first_activity,
    MAX(event_time) AS last_activity
FROM audit.activity_log
GROUP BY username
ORDER BY total_actions DESC;

-- Visão para atividades por tabela

```

```

CREATE OR REPLACE VIEW audit.table_activity AS
SELECT
    schema_name,
    table_name,
    COUNT(*) AS total_actions,
    COUNT(*) FILTER (WHERE action_type = 'INSERT') AS inserts,
    COUNT(*) FILTER (WHERE action_type = 'UPDATE') AS updates,
    COUNT(*) FILTER (WHERE action_type = 'DELETE') AS deletes,
    MIN(event_time) AS first_activity,
    MAX(event_time) AS last_activity
FROM audit.activity_log
GROUP BY schema_name, table_name
ORDER BY total_actions DESC;

-- Visão para alterações sensíveis
CREATE OR REPLACE VIEW audit.sensitive_changes AS
SELECT
    a.id,
    a.event_time,
    a.username,
    a.table_name,
    a.action_type,
    a.old_data,
    a.new_data
FROM audit.activity_log a
WHERE
    (a.table_name = 'clientes' AND
     (a.old_data->>'limite_credito' IS DISTINCT FROM a.new_data-
>>'limite_credito' OR
      a.old_data->>'dados_sensíveis' IS DISTINCT FROM a.new_data-
>>'dados_sensíveis'))
    OR
    (a.table_name = 'usuarios' AND a.action_type IN ('INSERT',
'DELETE'))
ORDER BY a.event_time DESC;

```

4.2 Funções para Relatórios

```
-- Função para relatório de atividades em um período
CREATE OR REPLACE FUNCTION audit.activity_report(
    start_time timestamp with time zone,
    end_time timestamp with time zone
)
RETURNS TABLE (
    username text,
    action_type text,
    table_name text,
    count bigint
) AS $$

BEGIN
    RETURN QUERY
    SELECT
        a.username,
        a.action_type,
        a.table_name,
        COUNT(*) AS count
    FROM audit.activity_log a
    WHERE a.event_time BETWEEN start_time AND end_time
    GROUP BY a.username, a.action_type, a.table_name
    ORDER BY count DESC;
END;

$$ LANGUAGE plpgsql;

-- Função para detectar atividades suspeitas
CREATE OR REPLACE FUNCTION audit.detect_suspicious_activity()
RETURNS TABLE (
    id integer,
    event_time timestamp with time zone,
    username text,
    action_type text,
    table_name text,
    suspicion_reason text
) AS $$
```

```

BEGIN
    RETURN QUERY

    -- Atividades fora do horário comercial
    SELECT
        a.id,
        a.event_time,
        a.username,
        a.action_type,
        a.table_name,
        'Atividade fora do horário comercial' AS suspicion_reason
    FROM audit.activity_log a
    WHERE
        EXTRACT(HOUR FROM a.event_time) < 8 OR
        EXTRACT(HOUR FROM a.event_time) > 18

    UNION ALL

    -- Volume anormal de operações
    SELECT
        a.id,
        a.event_time,
        a.username,
        a.action_type,
        a.table_name,
        'Volume anormal de operações' AS suspicion_reason
    FROM audit.activity_log a
    JOIN (
        SELECT
            username,
            table_name,
            COUNT(*) AS action_count
        FROM audit.activity_log
        WHERE event_time > CURRENT_TIMESTAMP - INTERVAL '1 hour'
        GROUP BY username, table_name
        HAVING COUNT(*) > 50
    ) high_volume ON a.username = high_volume.username AND
        a.table_name = high_volume.table_name

```

```

UNION ALL

-- Alterações em dados sensíveis
SELECT
    a.id,
    a.event_time,
    a.username,
    a.action_type,
    a.table_name,
    'Alteração em dados sensíveis' AS suspicion_reason
FROM audit.activity_log a
WHERE
    a.table_name = 'clientes' AND
    a.action_type = 'UPDATE' AND
    a.old_data->>'limite_credito' IS DISTINCT FROM a.new_data-
>>'limite_credito'

    ORDER BY event_time DESC;
END;
$$ LANGUAGE plpgsql;

```

Verificações e Testes

1. Gerar atividades para teste:

```

-- Inserir dados de teste
INSERT INTO lab_workspace.clientes (nome, email, segmento,
limite_credito)
VALUES ('Cliente Auditoria', 'auditoria@email.com', 'Varejo',
1000.00);

UPDATE lab_workspace.clientes
SET limite_credito = 5000.00
WHERE email = 'auditoria@email.com';

```

```
DELETE FROM lab_workspace.clientes
WHERE email = 'auditoria@email.com';

-- Simular login
SELECT lab_workspace.register_login('app_user');
```

2. Verificar logs de auditoria:

```
-- Verificar logs de atividade
SELECT * FROM audit.activity_log ORDER BY event_time DESC LIMIT 10;

-- Verificar logs de login
SELECT * FROM audit.login_attempts ORDER BY event_time DESC LIMIT 10;

-- Verificar logs de permissões
SELECT * FROM audit.permission_changes ORDER BY event_time DESC LIMIT 10;
```

3. Executar relatórios:

```
-- Relatório de atividades do dia
SELECT * FROM audit.activity_report(
    CURRENT_DATE,
    CURRENT_DATE + INTERVAL '1 day'
);

-- Verificar atividades suspeitas
SELECT * FROM audit.detect_suspicious_activity();

-- Verificar visões de resumo
SELECT * FROM audit.user_activity;
SELECT * FROM audit.table_activity;
SELECT * FROM audit.sensitive_changes;
```

Conclusão

Neste laboratório, você implementou:

- Um sistema abrangente de auditoria para banco de dados PostgreSQL
- Captura automática de alterações em tabelas via triggers
- Monitoramento de atividades sensíveis como alterações de permissões
- Relatórios e análises para identificação de atividades suspeitas

Estas técnicas são essenciais para:

1. **Conformidade regulatória:** Atender requisitos como GDPR, PCI-DSS, SOX, etc.
2. **Segurança:** Detectar e investigar atividades maliciosas ou não autorizadas
3. **Forense digital:** Reconstruir eventos em caso de incidentes de segurança
4. **Governança de dados:** Manter controle sobre quem acessa e modifica dados sensíveis

Próximos Passos

1. Implementar retenção e arquivamento de logs de auditoria
2. Configurar alertas automáticos para atividades suspeitas
3. Integrar com sistemas SIEM (Security Information and Event Management)
4. Implementar criptografia para dados de auditoria sensíveis



Nota: Registre a conclusão do laboratório:

```
UPDATE lab_workspace.lab_progress  
SET completion_time = CURRENT_TIMESTAMP, status =
```

```
'COMPLETED'  
WHERE lab_name = 'Auditoria de Segurança';
```

Laboratório: Migração de Permissões

Objetivo

Planejar e executar a migração de permissões entre ambientes de banco de dados, garantindo a consistência e segurança durante o processo de transição.

Cenário

Você é um DBA responsável por migrar um sistema de banco de dados de um ambiente de desenvolvimento para produção. Além da migração dos dados, você precisa garantir que todas as permissões sejam corretamente transferidas, mantendo a segurança e os controles de acesso.

Setup Inicial

```
-- Conectar ao banco de dados como superusuário
-- psql -U postgres -d lab_dcl

-- Criar schemas para simular ambientes
CREATE SCHEMA IF NOT EXISTS dev;
CREATE SCHEMA IF NOT EXISTS homolog;
CREATE SCHEMA IF NOT EXISTS prod;

-- Registrar início do laboratório
INSERT INTO lab_workspace.lab_progress (lab_name)
VALUES ('Migração de Permissões');
```

Parte 1: Preparação dos Ambientes

1.1 Criação de Estruturas nos Ambientes

```
-- Criar tabelas no ambiente de desenvolvimento
CREATE TABLE dev.clientes (
```

```

        id SERIAL PRIMARY KEY,
        nome VARCHAR(100) NOT NULL,
        email VARCHAR(100) UNIQUE,
        segmento VARCHAR(50),
        limite_credito DECIMAL(10, 2),
        data_cadastro TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    );

CREATE TABLE dev.produtos (
    id SERIAL PRIMARY KEY,
    codigo VARCHAR(20) UNIQUE,
    nome VARCHAR(100) NOT NULL,
    descricao TEXT,
    preco DECIMAL(10, 2) NOT NULL,
    estoque INTEGER DEFAULT 0,
    categoria VARCHAR(50)
);

CREATE TABLE dev_pedidos (
    id SERIAL PRIMARY KEY,
    cliente_id INTEGER REFERENCES dev.clientes(id),
    data_pedido TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    valor_total DECIMAL(10, 2),
    status VARCHAR(20) DEFAULT 'Pendente',
    observacoes TEXT
);

-- Criar tabelas no ambiente de homologação (estrutura idêntica)
CREATE TABLE homolog.clientes (LIKE dev.clientes INCLUDING ALL);
CREATE TABLE homolog.produtos (LIKE dev.produtos INCLUDING ALL);
CREATE TABLE homolog_pedidos (LIKE dev_pedidos INCLUDING ALL);
ALTER TABLE homolog_pedidos
    DROP CONSTRAINT IF EXISTS pedidos_cliente_id_fkey,
    ADD CONSTRAINT pedidos_cliente_id_fkey
    FOREIGN KEY (cliente_id) REFERENCES homolog.clientes(id);

-- Criar tabelas no ambiente de produção (estrutura idêntica)
CREATE TABLE prod.clientes (LIKE dev.clientes INCLUDING ALL);

```

```

CREATE TABLE prod.produtos (LIKE dev.produtos INCLUDING ALL);
CREATE TABLE prod.pedidos (LIKE dev.pedidos INCLUDING ALL);
ALTER TABLE prod.pedidos
    DROP CONSTRAINT IF EXISTS pedidos_cliente_id_fkey,
    ADD CONSTRAINT pedidos_cliente_id_fkey
    FOREIGN KEY (cliente_id) REFERENCES prod.clientes(id);

```

1.2 Configuração de Usuários e Roles

```

-- Criar roles para cada ambiente
CREATE ROLE dev_role;
CREATE ROLE homolog_role;
CREATE ROLE prod_role;

-- Criar roles funcionais
CREATE ROLE analista_role;
CREATE ROLE desenvolvedor_role;
CREATE ROLE dba_role;
CREATE ROLE suporte_role;

-- Criar usuários para cada ambiente
CREATE USER dev_user1 WITH PASSWORD 'dev123' IN ROLE
desenvolvedor_role, dev_role;
CREATE USER dev_user2 WITH PASSWORD 'dev123' IN ROLE
analista_role, dev_role;
CREATE USER homolog_user1 WITH PASSWORD 'hom123' IN ROLE
analista_role, homolog_role;
CREATE USER prod_user1 WITH PASSWORD 'prod123' IN ROLE
suporte_role, prod_role;
CREATE USER dba_user WITH PASSWORD 'dba123' IN ROLE dba_role;

-- Conceder permissões no ambiente de desenvolvimento
GRANT USAGE ON SCHEMA dev TO dev_role;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA dev TO
desenvolvedor_role;
GRANT SELECT, INSERT, UPDATE ON ALL TABLES IN SCHEMA dev TO
analista_role;

```

```
-- Conceder permissões no ambiente de homologação
GRANT USAGE ON SCHEMA homolog TO homolog_role;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA
homolog TO analista_role;

-- Conceder permissões no ambiente de produção
GRANT USAGE ON SCHEMA prod TO prod_role;
GRANT SELECT ON ALL TABLES IN SCHEMA prod TO suporte_role;

-- Conceder permissões administrativas
GRANT ALL PRIVILEGES ON SCHEMA dev, homolog, prod TO dba_role;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA dev, homolog, prod TO
dba_role;
```

Parte 2: Extração e Documentação de Permissões

2.1 Extração de Permissões Existentes

```
-- Criar tabela para documentar permissões
CREATE TABLE lab_workspace.permission_inventory (
    id SERIAL PRIMARY KEY,
    environment VARCHAR(20) NOT NULL,
    object_type VARCHAR(50) NOT NULL,
    object_name TEXT NOT NULL,
    grantee TEXT NOT NULL,
    privilege_type TEXT NOT NULL,
    is_grantable BOOLEAN,
    extracted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Função para extrair e documentar permissões
CREATE OR REPLACE FUNCTION lab_workspace.extract_permissions(env
text)
RETURNS void AS $$
BEGIN
    -- Limpar permissões anteriores do mesmo ambiente
```

```

DELETE FROM lab_workspace.permission_inventory
WHERE environment = env;

-- Inserir permissões de tabelas
INSERT INTO lab_workspace.permission_inventory (
    environment, object_type, object_name, grantee,
privilege_type, is_grantable
)
SELECT
    env AS environment,
    'TABLE' AS object_type,
    table_schema || '.' || table_name AS object_name,
    grantee,
    privilege_type,
    is_grantable::boolean
FROM information_schema.table_privileges
WHERE table_schema = env;

-- Inserir permissões de schemas
INSERT INTO lab_workspace.permission_inventory (
    environment, object_type, object_name, grantee,
privilege_type, is_grantable
)
SELECT
    env AS environment,
    'SCHEMA' AS object_type,
    schema_name AS object_name,
    grantee,
    privilege_type,
    is_grantable::boolean
FROM information_schema.usage_privileges
WHERE object_name = env;

-- Inserir permissões de colunas
INSERT INTO lab_workspace.permission_inventory (
    environment, object_type, object_name, grantee,
privilege_type, is_grantable
)

```

```

SELECT
    env AS environment,
    'COLUMN' AS object_type,
    table_schema || '.' || table_name || '.' || column_name AS
object_name,
    grantee,
    privilege_type,
    is_grantable::boolean
FROM information_schema.column_privileges
WHERE table_schema = env;
END;
$$ LANGUAGE plpgsql;

-- Extrair permissões de cada ambiente
SELECT lab_workspace.extract_permissions('dev');
SELECT lab_workspace.extract_permissions('homolog');
SELECT lab_workspace.extract_permissions('prod');

```

2.2 Análise de Permissões

```

-- Criar visão para análise de permissões
CREATE OR REPLACE VIEW lab_workspace.permission_analysis AS
WITH role_members AS (
    SELECT r.rolname AS role, u.username AS username
    FROM pg_roles r
    JOIN pg_auth_members m ON r.oid = m.roleid
    JOIN pg_roles u ON m.member = u.oid
    WHERE u.rolcanlogin

    UNION

    SELECT r.rolname AS role, r.rolname AS username
    FROM pg_roles r
    WHERE r.rolcanlogin
)
SELECT
    pi.environment,

```

```

pi.object_type,
pi.object_name,
pi.grantee,
pi.privilege_type,
array_agg(DISTINCT rm.username) AS affected_users
FROM lab_workspace.permission_inventory pi
LEFT JOIN role_members rm ON pi.grantee = rm.role
GROUP BY pi.environment, pi.object_type, pi.object_name,
pi.grantee, pi.privilege_type;

-- Consultar análise
SELECT * FROM lab_workspace.permission_analysis
ORDER BY environment, object_type, object_name;

```

Parte 3: Planejamento da Migração

3.1 Criação do Plano de Migração

```

-- Criar tabela para plano de migração
CREATE TABLE lab_workspace.migration_plan (
    id SERIAL PRIMARY KEY,
    source_environment VARCHAR(20) NOT NULL,
    target_environment VARCHAR(20) NOT NULL,
    object_type VARCHAR(50) NOT NULL,
    object_name TEXT NOT NULL,
    grantee TEXT NOT NULL,
    privilege_type TEXT NOT NULL,
    migration_status VARCHAR(20) DEFAULT 'Pending',
    migration_date TIMESTAMP,
    migration_notes TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Função para gerar plano de migração
CREATE OR REPLACE FUNCTION lab_workspace.generate_migration_plan(
    source_env text,
    target_env text
)
RETURNS TABLE(
    id int,
    source_environment VARCHAR(20),
    target_environment VARCHAR(20),
    object_type VARCHAR(50),
    object_name TEXT,
    grantee TEXT,
    privilege_type TEXT,
    migration_status VARCHAR(20),
    migration_date TIMESTAMP,
    migration_notes TEXT,
    created_at TIMESTAMP
) AS $$
BEGIN
    RETURN QUERY
    SELECT
        id,
        source_environment,
        target_environment,
        object_type,
        object_name,
        grantee,
        privilege_type,
        migration_status,
        migration_date,
        migration_notes,
        created_at
    FROM lab_workspace.migration_plan
    WHERE source_environment = source_env AND target_environment = target_env;
END;
$$ LANGUAGE plpgsql;

```

```

)
RETURNS void AS $$

BEGIN
    -- Limpar plano anterior para os mesmos ambientes
    DELETE FROM lab_workspace.migration_plan
    WHERE source_environment = source_env AND target_environment = target_env;

    -- Inserir permissões a serem migradas
    INSERT INTO lab_workspace.migration_plan (
        source_environment,
        target_environment,
        object_type,
        object_name,
        grantee,
        privilege_type,
        migration_notes
    )
    SELECT
        source_env,
        target_env,
        pi.object_type,
        -- Substituir o schema no nome do objeto
        REPLACE(pi.object_name, source_env || '.', target_env || '.')
        AS object_name,
        pi.grantee,
        pi.privilege_type,
        'Migrado automaticamente do plano'
    FROM lab_workspace.permission_inventory pi
    WHERE pi.environment = source_env;
END;

$$ LANGUAGE plpgsql;

-- Gerar plano de migração de dev para homolog
SELECT lab_workspace.generate_migration_plan('dev', 'homolog');

```

```
-- Gerar plano de migração de homolog para prod
SELECT lab_workspace.generate_migration_plan('homolog', 'prod');
```

3.2 Revisão e Ajuste do Plano

```
-- Ajustar plano para remover permissões sensíveis em produção
UPDATE lab_workspace.migration_plan
SET migration_status = 'Rejected',
    migration_notes = 'Permissão de DELETE não permitida em
produção'
WHERE target_environment = 'prod'
AND privilege_type = 'DELETE';

-- Ajustar plano para adicionar permissões específicas
INSERT INTO lab_workspace.migration_plan (
    source_environment,
    target_environment,
    object_type,
    object_name,
    grantee,
    privilege_type,
    migration_status,
    migration_notes
)
VALUES
('homolog', 'prod', 'TABLE', 'prod.clientes', 'suporte_role',
'SELECT', 'Pending', 'Permissão adicional para suporte'),
('homolog', 'prod', 'TABLE', 'prod_pedidos', 'suporte_role',
'SELECT', 'Pending', 'Permissão adicional para suporte');

-- Verificar plano final
SELECT * FROM lab_workspace.migration_plan
WHERE migration_status = 'Pending'
ORDER BY target_environment, object_type, object_name;
```

Parte 4: Execução da Migração

4.1 Geração de Scripts de Migração

```
-- Função para gerar script SQL de migração
CREATE OR REPLACE FUNCTION
lab_workspace.generate_migration_script(
    target_env text
)
RETURNS TABLE (
    script_order int,
    sql_command text
) AS $$

BEGIN
    RETURN QUERY

    -- Scripts para permissões de schema
    SELECT
        1 AS script_order,
        'GRANT ' || privilege_type || ' ON SCHEMA ' ||
        SPLIT_PART(object_name, '.', 1) || ' TO ' || grantee ||
    ';' AS sql_command
    FROM lab_workspace.migration_plan
    WHERE target_environment = target_env
    AND migration_status = 'Pending'
    AND object_type = 'SCHEMA'

    UNION ALL

    -- Scripts para permissões de tabela
    SELECT
        2 AS script_order,
        'GRANT ' || privilege_type || ' ON TABLE ' ||
        object_name || ' TO ' || grantee || ';' AS sql_command
    FROM lab_workspace.migration_plan
    WHERE target_environment = target_env
    AND migration_status = 'Pending'
    AND object_type = 'TABLE'
```

```

UNION ALL

-- Scripts para permissões de coluna
SELECT
    3 AS script_order,
    'GRANT ' || privilege_type || '(' ||
    SPLIT_PART(object_name, '.', 3) || ')' ON TABLE ' ||
    SPLIT_PART(object_name, '.', 1) || '.' ||
    SPLIT_PART(object_name, '.', 2) || ' TO ' || grantee ||
';' AS sql_command
FROM lab_workspace.migration_plan
WHERE target_environment = target_env
AND migration_status = 'Pending'
AND object_type = 'COLUMN'

ORDER BY script_order, sql_command;
END;
$$ LANGUAGE plpgsql;

-- Gerar script para homologação
SELECT * FROM lab_workspace.generate_migration_script('homolog');

-- Gerar script para produção
SELECT * FROM lab_workspace.generate_migration_script('prod');

```

4.2 Execução e Validação

```

-- Função para executar migração
CREATE OR REPLACE FUNCTION lab_workspace.execute_migration(
    target_env text
)
RETURNS TABLE (
    command_id int,
    sql_command text,
    execution_status text
) AS $$

DECLARE

```

```

cmd record;
cmd_count int := 0;
cmd_status text;

BEGIN
    -- Atualizar status para "Em Progresso"
    UPDATE lab_workspace.migration_plan
    SET migration_status = 'In Progress'
    WHERE target_environment = target_env
    AND migration_status = 'Pending';

    -- Executar cada comando
    FOR cmd IN SELECT * FROM
lab_workspace.generate_migration_script(target_env) LOOP
        cmd_count := cmd_count + 1;
        BEGIN
            EXECUTE cmd.sql_command;
            cmd_status := 'Success';

            command_id := cmd_count;
            sql_command := cmd.sql_command;
            execution_status := cmd_status;
            RETURN NEXT;
        EXCEPTION WHEN OTHERS THEN
            cmd_status := 'Error: ' || SQLERRM;

            command_id := cmd_count;
            sql_command := cmd.sql_command;
            execution_status := cmd_status;
            RETURN NEXT;
        END;
    END LOOP;

    -- Atualizar status para "Concluido"
    UPDATE lab_workspace.migration_plan
    SET
        migration_status = 'Completed',
        migration_date = CURRENT_TIMESTAMP
    WHERE target_environment = target_env

```

```

        AND migration_status = 'In Progress';

    RETURN;
END;
$$ LANGUAGE plpgsql;

-- Executar migração para homologação
SELECT * FROM lab_workspace.execute_migration('homolog');

-- Executar migração para produção
SELECT * FROM lab_workspace.execute_migration('prod');

```

Parte 5: Verificação e Documentação

5.1 Verificação de Permissões Migradas

```

-- Extrair permissões atualizadas
SELECT lab_workspace.extract_permissions('homolog');
SELECT lab_workspace.extract_permissions('prod');

-- Comparar permissões planejadas vs. implementadas
CREATE OR REPLACE VIEW lab_workspace.migration_verification AS
SELECT
    mp.target_environment,
    mp.object_type,
    mp.object_name,
    mp.grantee,
    mp.privilege_type,
    mp.migration_status,
    CASE
        WHEN pi.id IS NOT NULL THEN 'Verificado'
        ELSE 'Não Encontrado'
    END AS verification_status
FROM lab_workspace.migration_plan mp
LEFT JOIN lab_workspace.permission_inventory pi ON
    mp.target_environment = pi.environment AND
    mp.object_type = pi.object_type AND
    mp.privilege_type = pi.privilege_type AND
    mp.grantee = pi.grantee AND
    mp.object_name = pi.object_name

```

```

mp.object_name = pi.object_name AND
mp.grantee = pi.grantee AND
mp.privilege_type = pi.privilege_type
WHERE mp.migration_status = 'Completed';

-- Consultar verificação
SELECT * FROM lab_workspace.migration_verification
ORDER BY target_environment, verification_status DESC,
object_type, object_name;

```

5.2 Documentação Final

```

-- Criar relatório final de migração
CREATE OR REPLACE VIEW lab_workspace.migration_report AS
SELECT
    target_environment,
    COUNT(*) AS total_permissions,
    COUNT(*) FILTER (WHERE migration_status = 'Completed') AS
migrated_permissions,
    COUNT(*) FILTER (WHERE migration_status = 'Rejected') AS
rejected_permissions,
    COUNT(*) FILTER (WHERE migration_status = 'Completed' AND
verification_status = 'Verificado') AS verified_permissions,
    COUNT(*) FILTER (WHERE migration_status = 'Completed' AND
verification_status = 'Não Encontrado') AS missing_permissions
FROM lab_workspace.migration_verification
GROUP BY target_environment;

-- Consultar relatório
SELECT * FROM lab_workspace.migration_report;

```

Verificações e Testes

1. Verificar permissões como diferentes usuários:

```
psql -U dev_user1 -d lab_dcl  
psql -U homolog_user1 -d lab_dcl  
psql -U prod_user1 -d lab_dcl
```

2. Testar operações em cada ambiente:

```
-- Como dev_user1  
INSERT INTO dev.clientes (nome, email) VALUES ('Cliente Dev',  
'dev@email.com');  
  
-- Como homolog_user1  
INSERT INTO homolog.clientes (nome, email) VALUES ('Cliente  
Homolog', 'homolog@email.com');  
  
-- Como prod_user1  
SELECT * FROM prod.clientes;  
INSERT INTO prod.clientes (nome, email) VALUES ('Cliente Prod',  
'prod@email.com'); -- Deve falhar
```

3. Verificar inventário de permissões:

```
SELECT * FROM lab_workspace.permission_inventory  
WHERE environment = 'prod'  
ORDER BY object_type, object_name;
```

Conclusão

Neste laboratório, você implementou:

- Um processo estruturado para migração de permissões entre ambientes
- Ferramentas para extração e documentação de permissões existentes
- Mecanismos para planejamento e revisão de migrações
- Scripts automatizados para execução de migrações
- Métodos para verificação e validação de permissões migradas

Esta abordagem sistemática para migração de permissões oferece vários benefícios:

- 1. Consistência:** Garante que todas as permissões sejam corretamente transferidas entre ambientes
- 2. Segurança:** Mantém os controles de acesso e privilégios definidos
- 3. Eficiência:** Automatiza o processo de migração, reduzindo erros humanos
- 4. Documentação:** Fornece um registro detalhado das permissões em cada ambiente
- 5. Flexibilidade:** Permite ajustes e revisões no plano de migração antes da execução

Linguagem de Controle de Transação (TCL)

```
|| NEURAL.MATRIX >> TCL.FUNDAMENTOS
|| STATUS: ATIVO
|| SEGURANÇA: CRIPTOGRAFADO
|| ACESSO: CONHECIMENTO_PROFUNDO
```

ACID_QUEEN.PERSPECTIVA: Visão Geral

```
COMANDOS.CORE
|  
▶ BEGIN  
▶ COMMIT  
▶ ROLLBACK  
▶ SAVEPOINT
```

ARQUITETURA.TCL

NOSQL_PUNK.ALERTA: Conceitos Fundamentais

1. Transações

- Unidade lógica de trabalho
- Conjunto de operações indivisíveis
- Garantia de consistência

2. Propriedades ACID

- Atomicidade: tudo ou nada
- Consistência: integridade preservada
- Isolamento: transações independentes
- Durabilidade: mudanças permanentes

3. Estados da Transação

- Ativa: durante a execução
- Parcialmente confirmada: após última operação
- Confirmada: após COMMIT
- Falha: erro durante execução
- Abortada: após ROLLBACK

SEC_PHANTOM.DIRETRIZES: Boas Práticas

Transações Eficientes

- Mantenha transações curtas
- Minimize bloqueios
- Evite operações externas dentro de transações

Tratamento de Erros

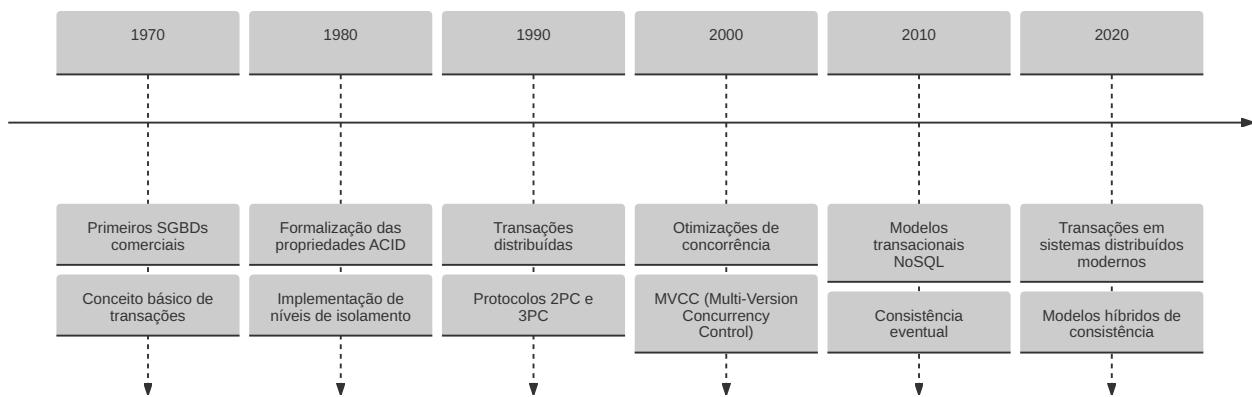
- Implemente tratamento de exceções
- Utilize SAVEPOINT para recuperação parcial
- Monitore deadlocks

Isolamento

- Escolha o nível adequado
- Entenda as anomalias de concorrência
- Considere o impacto no desempenho

TIME_LORD.CRONOLOGIA: Evolução Histórica

Evolução do Controle Transacional



BACKUP_PRIESTESS.RITUAL: Comandos Essenciais

BEGIN

Inicia uma transação:

```
BEGIN;  
-- OU  
BEGIN TRANSACTION;  
-- OU  
START TRANSACTION;
```

COMMIT

Confirma as alterações:

```
COMMIT;  
-- OU  
COMMIT TRANSACTION;
```

ROLLBACK

Desfaz as alterações:

```
ROLLBACK;  
-- OU  
ROLLBACK TRANSACTION;
```

SAVEPOINT

Cria um ponto de salvamento:

```
SAVEPOINT nome_savepoint;
```

ROLLBACK TO SAVEPOINT

Retorna a um ponto de salvamento:

```
ROLLBACK TO SAVEPOINT nome_savepoint;  
-- OU  
ROLLBACK TO nome_savepoint;
```

Navegação do Conteúdo

- Fundamentos de Transações ([Fundamentos de Transações](#))
- Commit e Rollback ([Commit e Rollback](#))
- Gerenciamento de Savepoints ([Gerenciamento de Savepoints](#))

Conclusão

O controle transacional é fundamental para garantir a integridade dos dados em sistemas de banco de dados. Através dos comandos TCL, os desenvolvedores podem gerenciar o

comportamento das transações, garantindo que as operações sejam executadas de forma atômica, consistente, isolada e durável.

A compreensão profunda dos mecanismos transacionais permite o desenvolvimento de aplicações robustas que mantêm a integridade dos dados mesmo em cenários de falha ou concorrência.

Fundamentos de Transações

Definição

Uma transação é uma unidade lógica de trabalho que contém uma ou mais operações de banco de dados. Todas as operações em uma transação são tratadas como uma única unidade atômica de trabalho que deve ser completamente executada ou completamente revertida.

Propriedades ACID

As transações são caracterizadas pelas propriedades ACID:

Atomicidade (Atomicity)

- Garante que todas as operações dentro da transação sejam tratadas como uma única unidade
- Ou todas as operações são executadas com sucesso, ou nenhuma é
- Não existem estados intermediários

```
BEGIN;  
    UPDATE contas SET saldo = saldo - 1000 WHERE id = 1;  
    UPDATE contas SET saldo = saldo + 1000 WHERE id = 2;  
COMMIT;
```

Se qualquer uma das operações falhar, nenhuma alteração será aplicada.

Consistência (Consistency)

- Garante que o banco de dados mude de um estado válido para outro estado válido
- Todas as regras de integridade são respeitadas
- Constraints, triggers e regras de negócio são preservadas

```
BEGIN;
    -- Esta operação viola uma constraint de saldo mínimo
    UPDATE contas SET saldo = -5000 WHERE id = 1;
COMMIT; -- Falha se houver uma constraint CHECK (saldo >= 0)
```

Isolamento (Isolation)

- Garante que as transações sejam isoladas umas das outras
- Uma transação não deve ser afetada por outras transações concorrentes
- Diferentes níveis de isolamento oferecem diferentes garantias

```
-- Transação 1
BEGIN;
    UPDATE produtos SET estoque = estoque - 1 WHERE id = 101;
    -- Outras operações...
COMMIT;

-- Transação 2 (concorrente)
BEGIN;
    SELECT estoque FROM produtos WHERE id = 101; -- 0 que será visto
depende do nível de isolamento
COMMIT;
```

Durabilidade (Durability)

- Garante que uma vez que uma transação seja confirmada, suas alterações são permanentes
- As alterações persistem mesmo em caso de falha do sistema
- Implementada através de logs de transação e mecanismos de recuperação

Níveis de Isolamento

Os SGBDs oferecem diferentes níveis de isolamento que equilibram consistência e desempenho:

Read Uncommitted

- Nível mais baixo de isolamento
- Permite leitura de dados não confirmados (dirty reads)
- Maior desempenho, menor consistência

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
BEGIN;
    -- Pode ler dados não confirmados de outras transações
    SELECT * FROM pedidos;
COMMIT;
```

Read Committed

- Evita leituras sujas (dirty reads)
- Permite leituras não repetíveis (non-repeatable reads)
- Nível padrão em muitos SGBDs

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN;
    -- Só lê dados confirmados, mas pode obter resultados diferentes
    -- em leituras repetidas
    SELECT * FROM pedidos;
    -- Algum processamento...
    SELECT * FROM pedidos; -- Pode retornar resultados diferentes
COMMIT;
```

Repeatable Read

- Evita leituras sujas e não repetíveis

- Permite leituras fantasmas (phantom reads)
- Maior consistência, menor desempenho

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;
    -- Mesmos dados serão lidos em consultas repetidas
    SELECT * FROM clientes WHERE saldo > 1000;
    -- Algun processamento...
    SELECT * FROM clientes WHERE saldo > 1000; -- Mesmos resultados
    que a primeira consulta
COMMIT;
```

Serializable

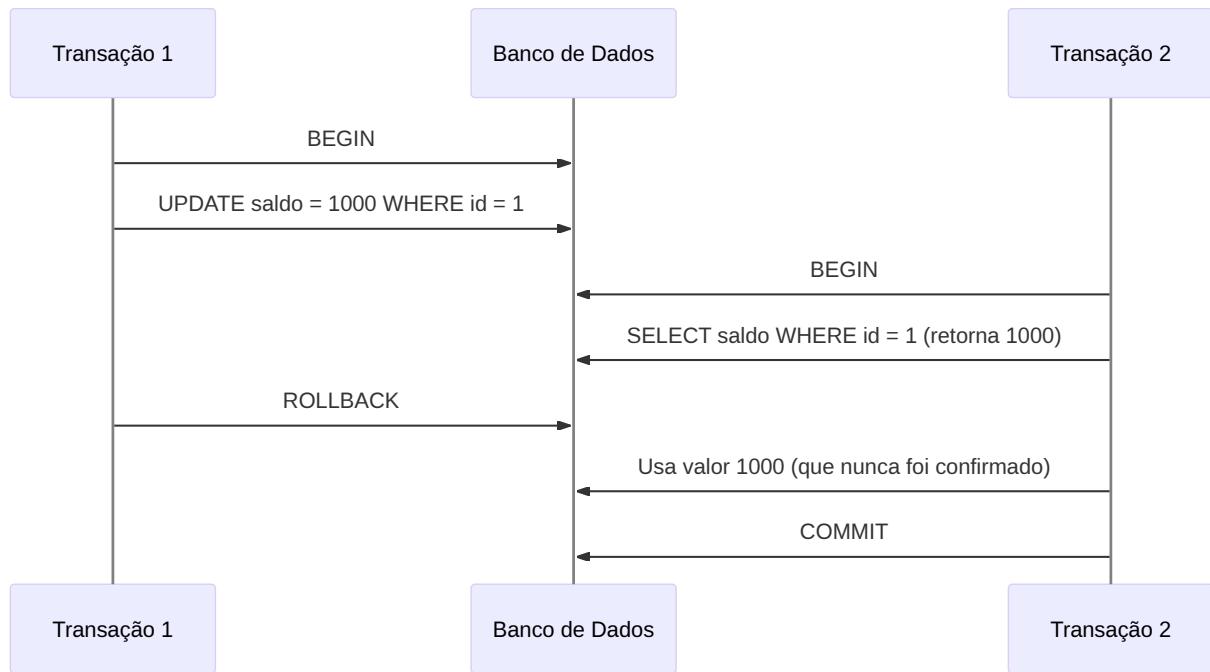
- Nível mais alto de isolamento
- Evita todos os problemas de concorrência
- Menor desempenho, maior consistência

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;
    -- Transações são completamente isoladas
    SELECT * FROM estoque WHERE produto_id = 101;
    -- Nenhuma outra transação pode modificar estes dados até o
    COMMIT
    COMMIT;
```

Anomalias de Concorrência

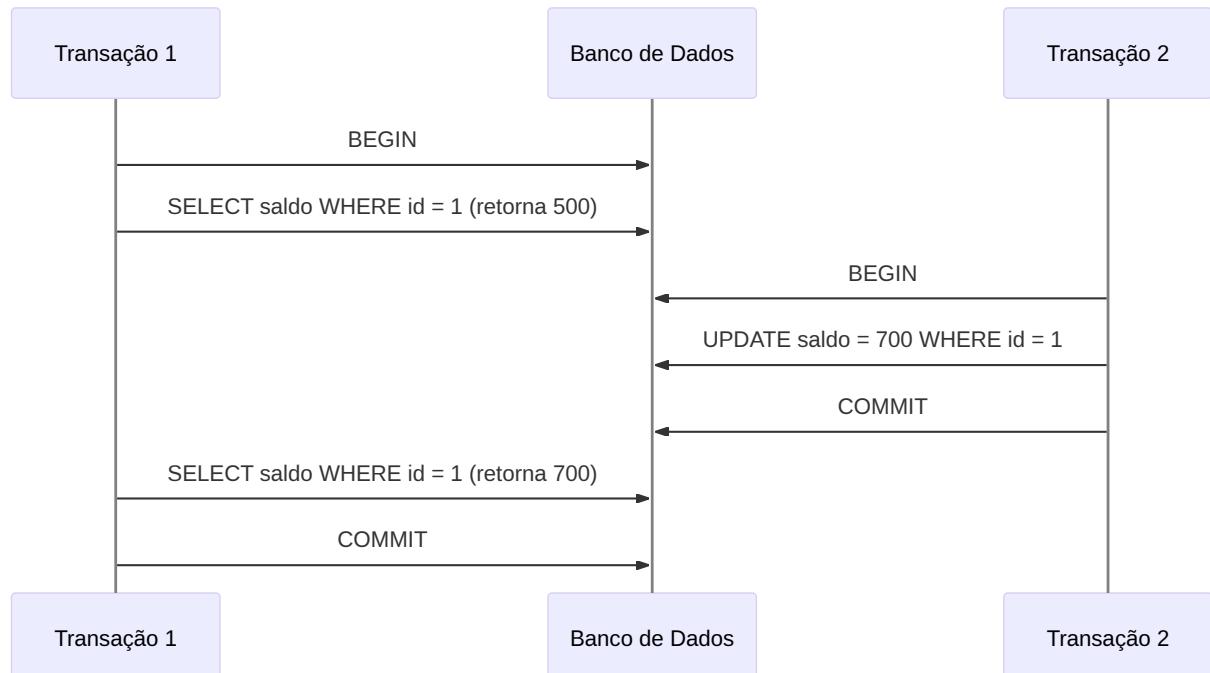
Dirty Read (Leitura Suja)

Ocorre quando uma transação lê dados que foram modificados por outra transação não confirmada.



Non-repeatable Read (Leitura Não Repetível)

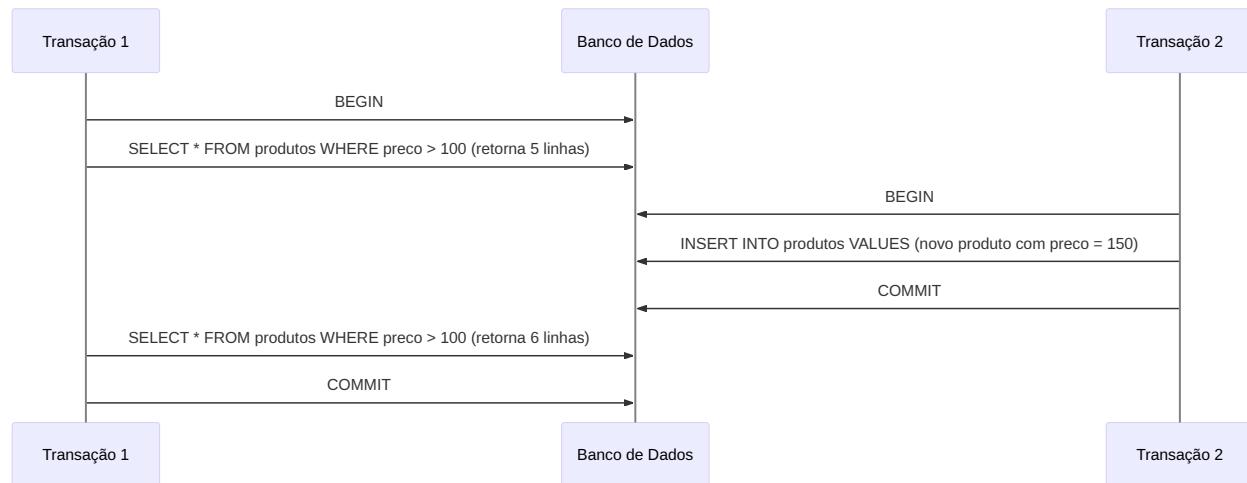
Ocorre quando uma transação relê dados e encontra que foram modificados por outra transação confirmada.



Phantom Read (Leitura Fantasma)

Ocorre quando uma transação relê um conjunto de linhas que satisfazem uma condição e

encontra que o conjunto mudou devido a outra transação.



Bloqueios (Locks)

Os SGBDs utilizam bloqueios para implementar o isolamento:

Tipos de Bloqueios

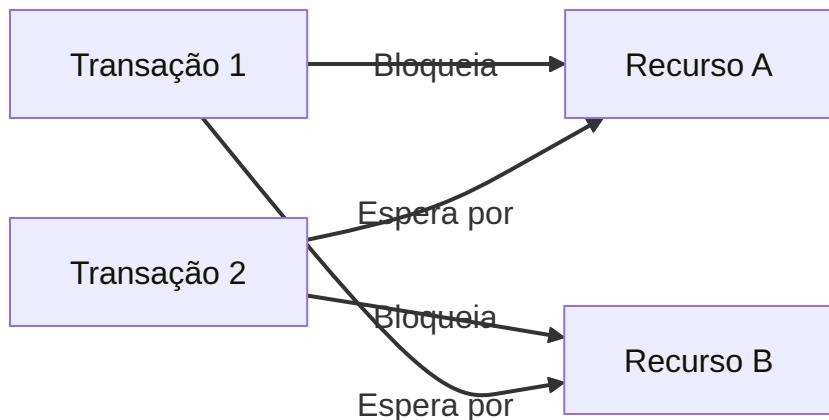
- **Compartilhado (Shared)**: Permite leitura, mas não escrita
- **Exclusivo (Exclusive)**: Permite leitura e escrita, bloqueia outros acessos
- **Atualização (Update)**: Inicialmente compartilhado, pode ser promovido para exclusivo
- **Intenção (Intent)**: Indica intenção de adquirir bloqueios em níveis inferiores

Granularidade de Bloqueios

- **Banco de dados**: Bloqueia todo o banco
- **Tabela**: Bloqueia uma tabela inteira
- **Página**: Bloqueia uma página de dados
- **Linha**: Bloqueia apenas uma linha
- **Coluna**: Bloqueia apenas uma coluna

Deadlocks

Situação onde duas ou mais transações estão esperando uma pela outra para liberar recursos.



Controle de Concorrência Multiversão (MVCC)

Muitos SGBDs modernos utilizam MVCC (Multi-Version Concurrency Control):

- Mantém múltiplas versões dos dados
- Leitores não bloqueiam escritores
- Escritores não bloqueiam leitores
- Cada transação vê um snapshot consistente do banco de dados
- Implementado em PostgreSQL, Oracle, MySQL InnoDB, etc.

```
Linha 1, versão 1 (txid 100): {id: 1, nome: "João", saldo: 500}
Linha 1, versão 2 (txid 105): {id: 1, nome: "João", saldo: 700}
```

Exemplos Práticos

Transferência Bancária

```
BEGIN;
    -- Verificar saldo
    SELECT saldo FROM contas WHERE id = 101;
```

```

-- Debitar da conta origem
UPDATE contas SET saldo = saldo - 1000 WHERE id = 101;

-- Creditar na conta destino
UPDATE contas SET saldo = saldo + 1000 WHERE id = 102;

-- Registrar a transferência
INSERT INTO transferencias (origem, destino, valor, data)
VALUES (101, 102, 1000, CURRENT_TIMESTAMP);

-- Confirmar todas as operações
COMMIT;

```

Processamento de Pedido

```

BEGIN;
-- Inserir pedido
INSERT INTO pedidos (cliente_id, data, valor_total)
VALUES (201, CURRENT_DATE, 0)
RETURNING id INTO v_pedido_id;

-- Inserir itens do pedido
INSERT INTO itens_pedido (pedido_id, produto_id, quantidade,
preco_unitario)
VALUES
(v_pedido_id, 301, 2, 29.90),
(v_pedido_id, 302, 1, 49.90);

-- Atualizar estoque
UPDATE produtos SET estoque = estoque - 2 WHERE id = 301;
UPDATE produtos SET estoque = estoque - 1 WHERE id = 302;

-- Calcular valor total
UPDATE pedidos
SET valor_total = (
  SELECT SUM(quantidade * preco_unitario)
  FROM itens_pedido

```

```
    WHERE pedido_id = v_pedido_id
)
WHERE id = v_pedido_id;

-- Confirmar todas as operações
COMMIT;
```

Conclusão

As transações são fundamentais para garantir a integridade dos dados em sistemas de banco de dados. Compreender seus princípios e mecanismos é essencial para desenvolver aplicações robustas e confiáveis.

Os conceitos ACID, níveis de isolamento e controle de concorrência são pilares do processamento transacional que permitem que múltiplos usuários trabalhem simultaneamente com os dados, mantendo a consistência e a integridade do banco de dados.

Commit e Rollback

Visão Geral

Os comandos `COMMIT` e `ROLLBACK` são fundamentais para o controle transacional em bancos de dados. Eles determinam o destino final das operações realizadas dentro de uma transação.

Comando COMMIT

O comando `COMMIT` finaliza a transação atual e torna permanentes todas as alterações realizadas desde o início da transação.

Sintaxe

```
COMMIT [TRANSACTION];
-- ou simplesmente
COMMIT;
```

Efeitos do COMMIT

- Persistência:** Todas as alterações se tornam permanentes no banco de dados
- Visibilidade:** As alterações se tornam visíveis para outras transações
- Liberação de Bloqueios:** Todos os bloqueios adquiridos pela transação são liberados
- Pontos de Salvamento:** Todos os savepoints criados na transação são descartados
- Finalização:** A transação é encerrada e uma nova transação é iniciada automaticamente em alguns SGBDs

Exemplos de COMMIT

Exemplo Básico

```
BEGIN;
INSERT INTO clientes (nome, email)
VALUES ('Maria Silva', 'maria@email.com');
```

```
UPDATE produtos
SET estoque = estoque - 1
WHERE id = 101;
COMMIT;
```

Commit Condicional

```
BEGIN;
    UPDATE contas
    SET saldo = saldo - 1000
    WHERE id = 1;

    UPDATE contas
    SET saldo = saldo + 1000
    WHERE id = 2;

    -- Verificar se ambas as contas existem
    IF (SELECT COUNT(*) FROM contas WHERE id IN (1, 2)) = 2 THEN
        COMMIT;
    ELSE
        ROLLBACK;
    END IF;
```

Comando ROLLBACK

O comando **ROLLBACK** desfaz todas as alterações realizadas desde o início da transação ou desde um ponto de salvamento específico.

Sintaxe

```
ROLLBACK [TRANSACTION];
-- ou simplesmente
ROLLBACK;
```

```
-- Para rollback parcial até um savepoint  
ROLLBACK [TRANSACTION] TO [SAVEPOINT] savepoint_name;
```

Efeitos do ROLLBACK

1. **Reversão:** Todas as alterações são desfeitas
2. **Liberação de Bloqueios:** Todos os bloqueios adquiridos pela transação são liberados
3. **Pontos de Salvamento:** Todos os savepoints são descartados (exceto em rollback parcial)
4. **Finalização:** A transação é encerrada e uma nova transação é iniciada automaticamente em alguns SGBDs

Exemplos de ROLLBACK

Rollback Completo

```
BEGIN;  
    DELETE FROM pedidos WHERE cliente_id = 101;  
  
    -- Ops, não era para excluir todos os pedidos!  
    ROLLBACK;
```

Rollback com Tratamento de Erros

```
BEGIN;  
    DECLARE EXIT HANDLER FOR SQLEXCEPTION  
    BEGIN  
        ROLLBACK;  
        SELECT 'Transação abortada devido a erro' AS mensagem;  
    END;  
  
    UPDATE produtos SET preco = preco * 1.1;  
    INSERT INTO log_alteracoes (tabela, descricao)  
    VALUES ('produtos', 'Aumento de 10% nos preços');
```

```
    COMMIT;  
END;
```

Commit vs. Rollback: Quando Usar

Use COMMIT quando:

- 1. Operações Bem-Sucedidas:** Todas as operações foram concluídas com sucesso
- 2. Dados Válidos:** Os dados estão em um estado consistente
- 3. Ponto de Finalização:** Você atingiu um ponto lógico de conclusão no processamento

Use ROLLBACK quando:

- 1. Erros Detectados:** Ocorreram erros durante o processamento
- 2. Dados Inválidos:** Os dados não atendem às regras de negócio ou restrições
- 3. Operação Cancelada:** O usuário ou o sistema decidiu cancelar a operação
- 4. Teste ou Simulação:** Você está apenas testando ou simulando operações

Auto-Commit

Muitos SGBDs e ferramentas de cliente possuem um modo "auto-commit" que automaticamente confirma cada instrução SQL individual como uma transação separada.

Habilitando/Desabilitando Auto-Commit

```
-- PostgreSQL  
SET autocommit = ON; -- ou OFF  
  
-- MySQL  
SET autocommit = 1; -- ou 0  
  
-- SQL Server  
SET IMPLICIT_TRANSACTIONS OFF; -- auto-commit ON
```

```
SET IMPLICIT_TRANSACTIONS ON; -- auto-commit OFF  
  
-- Oracle  
-- Sempre requer COMMIT explícito, exceto para DDL
```

Implicações do Auto-Commit

- **Vantagens:** Simplicidade, menor risco de deixar transações abertas
- **Desvantagens:** Perda de atomicidade para operações múltiplas, potencial redução de desempenho

Commit Implícito

Algumas operações podem causar um commit implícito, mesmo dentro de uma transação explícita:

1. **Comandos DDL:** CREATE, ALTER, DROP, etc.
2. **Comandos DCL:** GRANT, REVOKE, etc.
3. **Comandos de Administração:** ANALYZE, VACUUM, etc.
4. **Conexão/Desconexão:** Dependendo do SGBD e configuração

```
BEGIN;  
    INSERT INTO tabela1 VALUES (1, 'teste');  
  
    -- Em muitos SGBDs, isto causa um commit implícito  
    CREATE INDEX idx_tabela1 ON tabela1(coluna1);  
  
    -- Esta operação já estará em uma nova transação  
    UPDATE tabela2 SET coluna1 = 'valor';  
COMMIT;
```

Práticas Recomendadas

1. Transações Curtas e Focadas

```
-- Preferível: Transações pequenas e focadas
BEGIN;
    UPDATE estoque SET quantidade = quantidade - 10 WHERE
produto_id = 101;
    INSERT INTO movimentos (produto_id, quantidade, tipo) VALUES
(101, 10, 'SAÍDA');
COMMIT;

BEGIN;
    -- Outra operação lógica separada
    UPDATE faturamento SET total = total + 1000 WHERE mes =
CURRENT_MONTH;
COMMIT;
```

2. Tratamento Adequado de Erros

```
BEGIN;
    SAVEPOINT antes_operacoes_criticas;

    -- Tente executar operações críticas
    UPDATE contas SET saldo = saldo - 5000 WHERE id = 101;

    -- Verifique se a conta tem saldo suficiente
    IF (SELECT saldo FROM contas WHERE id = 101) < 0 THEN
        -- Desfaça apenas as operações críticas
        ROLLBACK TO antes_operacoes_criticas;
        -- Continue com outras operações
    ELSE
        -- Prossiga com o restante da transação
    END IF;

    -- Operações não críticas
    INSERT INTO log_atividades (descricao) VALUES ('Tentativa de
saque');
```

```
COMMIT;
```

3. Evite Transações de Longa Duração

```
-- Evite:  
BEGIN;  
    -- Processamento em lote de milhões de registros  
    UPDATE grande_tabela SET status = 'PROCESSADO' WHERE status =  
    'PENDENTE';  
    -- Pode levar muito tempo e bloquear outros processos  
COMMIT;  
  
-- Prefira:  
DO $$  
DECLARE  
    batch_size INT := 1000;  
    total_updated INT := 0;  
BEGIN  
    LOOP  
        BEGIN  
            UPDATE grande_tabela  
            SET status = 'PROCESSADO'  
            WHERE id IN (  
                SELECT id  
                FROM grande_tabela  
                WHERE status = 'PENDENTE'  
                LIMIT batch_size  
            );  
            GET DIAGNOSTICS total_updated = ROW_COUNT;  
            EXIT WHEN total_updated = 0;  
        COMMIT;  
    END;
```

```
END LOOP;  
END $$;
```

Conclusão

Os comandos `COMMIT` e `ROLLBACK` são ferramentas essenciais para garantir a integridade dos dados em sistemas de banco de dados. Eles permitem que os desenvolvedores controlem precisamente quando as alterações devem ser aplicadas permanentemente ou descartadas.

O uso adequado desses comandos, combinado com uma estratégia bem planejada de transações, é fundamental para desenvolver aplicações robustas que mantenham a consistência dos dados mesmo em cenários de falha ou concorrência.

Gerenciamento de Savepoints

Introdução aos Savepoints

Savepoints são marcadores dentro de uma transação que permitem o rollback parcial, desfazendo apenas parte das operações realizadas, sem precisar abortar toda a transação.

Comandos Básicos

Criação de Savepoint

```
SAVEPOINT nome_do_savepoint;
```

Rollback para um Savepoint

```
ROLLBACK TO [SAVEPOINT] nome_do_savepoint;  
-- ou em alguns SGBDs  
ROLLBACK TRANSACTION TO [SAVEPOINT] nome_do_savepoint;
```

Liberação de Savepoint

Em alguns SGBDs, é possível liberar um savepoint que não é mais necessário:

```
RELEASE SAVEPOINT nome_do_savepoint;
```

Comportamento dos Savepoints

Ciclo de Vida

- **Criação:** Um savepoint é criado com o comando `SAVEPOINT`
- **Uso:** Um savepoint pode ser usado como destino de um `ROLLBACK TO`
- **Liberação:** Um savepoint pode ser liberado explicitamente com `RELEASE SAVEPOINT`

- **Descarte:** Todos os savepoints são descartados quando a transação é finalizada com `COMMIT` ou `ROLLBACK`

Aninhamento

Savepoints podem ser aninhados, permitindo níveis de granularidade no controle de transações:

```
BEGIN;  
    INSERT INTO tabela1 VALUES (1);  
  
    SAVEPOINT sp1;  
    UPDATE tabela2 SET coluna1 = 'valor1';  
  
    SAVEPOINT sp2;  
    DELETE FROM tabela3 WHERE id = 5;  
  
    -- Desfaz apenas o DELETE  
    ROLLBACK TO sp2;  
  
    -- Continua a transação  
    INSERT INTO tabela4 VALUES (2);  
  
    -- Desfaz o UPDATE e o INSERT em tabela4  
    ROLLBACK TO sp1;  
  
    -- Continua a transação  
    UPDATE tabela5 SET coluna1 = 'valor2';  
COMMIT;
```

Comportamento em Diferentes SGBDs

PostgreSQL

- Suporta savepoints aninhados
- Permite `RELEASE SAVEPOINT`
- Savepoints persistem após `ROLLBACK TO`

```
BEGIN;
    INSERT INTO tabela VALUES (1);
    SAVEPOINT sp1;
    INSERT INTO tabela VALUES (2);
    SAVEPOINT sp2;
    INSERT INTO tabela VALUES (3);
    ROLLBACK TO sp1;
    -- Neste ponto, sp2 ainda existe, mas é inútil
    INSERT INTO tabela VALUES (4);
COMMIT;
-- Resultado: tabela contém valores 1 e 4
```

MySQL

- Suporta savepoints aninhados
- Permite RELEASE SAVEPOINT
- Comportamento similar ao PostgreSQL

Oracle

- Suporta savepoints aninhados
- Permite ROLLBACK TO sem a palavra SAVEPOINT
- Não mantém savepoints após um rollback para um savepoint anterior

```
BEGIN;
    INSERT INTO tabela VALUES (1);
    SAVEPOINT sp1;
    INSERT INTO tabela VALUES (2);
    SAVEPOINT sp2;
    INSERT INTO tabela VALUES (3);
    ROLLBACK TO sp1;
```

```
-- Em Oracle, sp2 não existe mais neste ponto  
COMMIT;
```

SQL Server

- Suporta savepoints
- Usa `SAVE TRANSACTION` em vez de `SAVEPOINT`
- Não suporta `RELEASE SAVEPOINT`

```
BEGIN TRANSACTION;  
    INSERT INTO tabela VALUES (1);  
    SAVE TRANSACTION sp1;  
    INSERT INTO tabela VALUES (2);  
    ROLLBACK TRANSACTION sp1;  
    INSERT INTO tabela VALUES (3);  
COMMIT;  
-- Resultado: tabela contém valores 1 e 3
```

Casos de Uso

1. Processamento em Etapas

Útil quando uma transação tem várias etapas lógicas e você deseja poder reverter para o início de qualquer etapa.

```
BEGIN;  
    -- Etapa 1: Criar pedido  
    INSERT INTO pedidos (cliente_id, data)  
    VALUES (101, CURRENT_DATE)  
    RETURNING id INTO v_pedido_id;  
  
    SAVEPOINT apos_criar_pedido;  
  
    -- Etapa 2: Adicionar itens  
    INSERT INTO itens_pedido (pedido_id, produto_id, quantidade,
```

```

    preco)
VALUES
    (v_pedido_id, 201, 2, 29.90),
    (v_pedido_id, 202, 1, 49.90);

SAVEPOINT apos_adicionar_itens;

-- Etapa 3: Atualizar estoque
UPDATE produtos SET estoque = estoque - 2 WHERE id = 201;
UPDATE produtos SET estoque = estoque - 1 WHERE id = 202;

-- Verificar se há estoque suficiente
IF EXISTS (SELECT 1 FROM produtos WHERE id IN (201, 202) AND
estoque < 0) THEN
    -- Desfazer apenas a atualização de estoque
    ROLLBACK TO apos_adicionar_itens;
    -- Marcar pedido como "aguardando estoque"
    UPDATE pedidos SET status = 'AGUARDANDO_ESTOQUE' WHERE id
= v_pedido_id;
END IF;

-- Finalizar transação
COMMIT;

```

2. Tratamento de Erros

Permite implementar lógica de recuperação de erros sem abortar toda a transação.

```

BEGIN;
    -- Operações principais
    INSERT INTO clientes (nome, email) VALUES ('João Silva',
'joao@email.com');

    SAVEPOINT apos_cliente;

    -- Operações que podem falhar
    BEGIN

```

```

-- Tenta inserir endereço
INSERT INTO enderecos (cliente_id, cep, logradouro)
VALUES (LASTVAL(), '12345-678', 'Rua das Flores, 123');
EXCEPTION WHEN OTHERS THEN
    -- Se falhar, volta para depois de inserir o cliente
    ROLLBACK TO apos_cliente;
    -- Registra o problema
    INSERT INTO log_erros (operacao, mensagem)
    VALUES ('inserir_endereco', SQLERRM);
END;

-- Continua com outras operações
INSERT INTO contatos (cliente_id, telefone)
VALUES (LASTVAL(), '(11) 98765-4321');

COMMIT;

```

3. Validação de Dados em Etapas

Permite validar dados em etapas, revertendo apenas as etapas com problemas.

```

BEGIN;
    -- Etapa 1: Importar dados de clientes
    INSERT INTO clientes_temp
    SELECT * FROM arquivo_importacao_clientes;

    SAVEPOINT apos_importar_clientes;

    -- Etapa 2: Validar dados de clientes
    DELETE FROM clientes_temp
    WHERE email IS NULL OR email NOT LIKE '%@%.%';

    -- Verificar se muitos registros foram removidos
    IF (SELECT COUNT(*) FROM clientes_temp) <
        (SELECT COUNT(*) * 0.9 FROM arquivo_importacao_clientes)
    THEN
        -- Muitos registros inválidos, voltar e tentar corrigir

```

```

ROLLBACK TO apos_importar_clientes;

-- Tentar corrigir emails
UPDATE clientes_temp
SET email = nome || '@dominio.com'
WHERE email IS NULL OR email NOT LIKE '%@%.%';
END IF;

SAVEPOINT apos_validar_clientes;

-- Etapa 3: Mover para tabela definitiva
INSERT INTO clientes
SELECT * FROM clientes_temp;

COMMIT;

```

Práticas Recomendadas

1. Nomeação Clara e Consistente

Use nomes descritivos que indiquem o estado da transação após cada savepoint:

```

-- Bom: nomes descritivos
SAVEPOINT apos_criar_cliente;
SAVEPOINT apos_processar_pagamento;
SAVEPOINT apos_atualizar_estoque;

-- Evite: nomes genéricos ou numéricos
SAVEPOINT sp1;
SAVEPOINT ponto2;
SAVEPOINT x123;

```