Apostila de SQLite – Nível Básico

Prof. Adriano Tech

Site para acesso gratuito: https://sqliteonline.com/

Este material tem como objetivo apresentar os conceitos básicos da linguagem SQL (Structured Query Language). Esta linguagem tem grande aplicabilidade, uma vez que é um dos mais populares Sistema Gerenciador de Banco de Dados (SGBD).

Os sistemas gerenciadores de banco de dados desempenham um papel importante no desenvolvimento de sistemas voltados para a Zootecnia de Precisão e para o manejo de fazendas e linhas de produção. Com o avanço da tecnologia e a crescente quantidade de dados gerados no setor agropecuário, o uso de bancos de dados se tornou essencial para armazenar, gerenciar e extrair informações valiosas que impulsionam a tomada de decisões e a eficiência nas atividades agropecuárias.

Importância dos sistemas gerenciadores de banco de dados podem ser melhor compreendidas quando entendemos as suas áreas de aplicação, como:

Armazenamento de dados: Os sistemas gerenciadores de banco de dados permitem armazenar grandes volumes de dados relacionados à Zootecnia de Precisão e ao manejo de fazendas, incluindo informações sobre animais, genética, alimentação, saúde, produção e muito mais. Esses dados são essenciais para monitorar o desempenho dos animais, rastrear históricos, identificar padrões e tendências, e fornecer insights valiosos para melhorar a eficiência e a produtividade.

Gerenciamento de dados: Com um sistema gerenciador de banco de dados adequado, é possível organizar os dados de forma estruturada, definir relacionamentos entre as informações, garantir a integridade dos

dados e aplicar restrições para manter a consistência dos registros. Isso permite uma gestão eficiente das informações, facilitando a recuperação, a atualização e a exclusão de dados conforme necessário.

Recuperação de informações: Os sistemas gerenciadores de banco de dados permitem realizar consultas complexas para recuperar informações específicas. Com o uso de consultas SQL, é possível extrair dados relevantes com base em critérios específicos, como o desempenho de determinado animal, a produtividade de uma fazenda em um determinado período, entre outros. Essas informações são fundamentais para monitorar o progresso, identificar problemas e tomar decisões embasadas em dados concretos.

Análise de dados: Com um banco de dados bem estruturado e o suporte de sistemas gerenciadores de banco de dados, é possível realizar análises avançadas dos dados agropecuários. Essas análises podem incluir cálculos de indicadores de desempenho, tendências de produção, correlações entre fatores ambientais e produtivos, modelos de previsão, entre outros. Essas análises fornecem insights valiosos para otimizar processos, melhorar a eficiência e maximizar os resultados nas atividades agropecuárias.

Assim, os SGBD são fundamentais para o desenvolvimento de sistemas que desempenham um papel crucial no armazenamento, gerenciamento e recuperação de dados, permitindo uma gestão eficiente das informações e fornecendo insights valiosos para a tomada de decisões e aprimoramento das práticas agropecuárias. Com a utilização adequada dos sistemas gerenciadores de banco de dados, é possível impulsionar a produtividade, a sustentabilidade e o sucesso no setor agropecuário.

Aqui estão alguns exemplos de SGBD amplamente utilizados:

1. **MySQL:** Um sistema gerenciador de banco de dados relacional de código aberto amplamente utilizado. É conhecido por sua confiabilidade, desempenho e facilidade de uso.
2. **Oracle Database:** Um sistema gerenciador de banco de dados relacional líder de mercado, conhecido por sua escalabilidade, segurança e recursos avançados.
3. **Microsoft SQL Server:** Um sistema gerenciador de banco de dados relacional desenvolvido pela Microsoft, que oferece recursos avançados, escalabilidade e integração com outras tecnologias da Microsoft.
4. **PostgreSQL**: Um sistema gerenciador de banco de dados relacional de código aberto conhecido por sua robustez, conformidade com padrões, recursos avançados e comunidade ativa.
5. **MongoDB:** Um sistema gerenciador de banco de dados NoSQL orientado a documentos, que oferece alta flexibilidade, escalabilidade e facilidade de uso. É especialmente adequado para dados semiestruturados e de natureza variável.
6. **Redis:** Um sistema gerenciador de banco de dados NoSQL em memória, projetado para alta velocidade e baixa latência. É amplamente utilizado para armazenar dados em cache, sessões e filas de mensagens.
7. **Cassandra:** Um sistema gerenciador de banco de dados NoSQL distribuído, projetado para escalabilidade horizontal e alta disponibilidade. É especialmente adequado para aplicativos com grandes volumes de dados distribuídos em vários nós.
8. **SQLite:** Um sistema gerenciador de banco de dados leve e incorporado, que não requer um servidor separado e é adequado para aplicativos de desktop e dispositivos móveis.

Esses são apenas alguns exemplos de sistemas gerenciadores de banco de dados, cada um com suas características e casos de uso específicos. A escolha do sistema adequado depende dos requisitos do projeto, do volume de dados,

da escalabilidade necessária, da integração com outras tecnologias e dos recursos específicos desejados.

Neste livro vamos dar uma introdução a linguagem MS SQL / SQLite.

1. O que é a linguagem SQL?

Desta maneira, pode-se definir SQL (Structured Query Language) como uma linguagem de programação específica para bancos de dados relacionais, sendo utilizada para criar, modificar e manipular bancos de dados, bem como para recuperar e gerenciar os dados armazenados em suas tabelas.

O SQL permite realizar uma variedade de tarefas relacionadas a bancos de dados, como criar tabelas, definir relacionamentos entre tabelas, inserir, atualizar e excluir registros, realizar consultas complexas para recuperar informações específicas e executar operações de agregação e cálculos.

A linguagem SQL é projetada com uma sintaxe declarativa, ou seja, você especifica o que deseja obter, mas não precisa se preocupar com os detalhes de como isso é feito internamente. Isso permite que os desenvolvedores se concentrem nos requisitos e na lógica dos dados, deixando para o sistema de gerenciamento de banco de dados a responsabilidade de executar as operações de forma eficiente.

O SQL é amplamente utilizado em sistemas de gerenciamento de bancos de dados relacionais, como MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server, SQLite, entre outros. Ele se tornou um padrão da indústria e é amplamente adotado por desenvolvedores, administradores de banco de dados e profissionais de dados para trabalhar com bancos de dados relacionais.

Por meio do SQL pode-se manipular banco de dados (BD) e, suas respectivas tabelas. Assim, um banco de dados é uma coleção organizada de dados estruturados que são armazenados eletronicamente em um sistema de computador. Os bancos de dados são projetados para armazenar, gerenciar e recuperar informações de forma eficiente e confiável.

Existem vários tipos de BD, cada um adequado para diferentes necessidades e casos de uso. Aqui estão alguns exemplos dos tipos de bancos de dados mais comuns:

1. **Banco de Dados Relacional:** É o tipo mais comum de banco de dados. Utiliza a estrutura relacional, baseada em tabelas, onde os dados são armazenados em linhas e colunas. Exemplos populares de bancos de dados relacionais incluem o MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server.
2. **Banco de Dados NoSQL:** Diferente dos bancos de dados relacionais, os bancos de dados NoSQL (Not Only SQL) não seguem uma estrutura tabular rígida. Eles são projetados para lidar com grandes volumes de dados não estruturados ou semiestruturados. Alguns exemplos de bancos de dados NoSQL são MongoDB, Cassandra, Redis.
3. **Banco de Dados de Grafos:** Esse tipo de banco de dados é projetado para representar e armazenar relacionamentos complexos entre os dados. Ele utiliza uma estrutura de grafo, onde os nós representam entidades e as arestas representam os relacionamentos entre essas entidades. Exemplos de bancos de dados de grafos incluem Neo4j, Amazon Neptune.
4. **Banco de Dados em Memória:** Esses bancos de dados são otimizados para armazenar e recuperar dados diretamente da memória do computador, em vez de usar armazenamento em disco. Eles oferecem alta velocidade de acesso aos dados, sendo ideais para aplicativos que exigem baixa latência. Exemplos incluem Redis, Memcached.
5. **Banco de Dados Orientado a Documentos:** Esses bancos de dados armazenam e recuperam dados no formato de documentos, geralmente usando formatos como JSON ou XML. Eles são flexíveis e escaláveis, permitindo a adição de campos sem alterar a estrutura geral do documento. MongoDB é um exemplo popular de banco de dados orientado a documentos.
6. **Banco de Dados em Nuvem:** Esses bancos de dados são executados em servidores em nuvem e oferecem escalabilidade e flexibilidade. Eles podem ser baseados em modelos de banco de dados relacionais ou NoSQL. Exemplos incluem Google Cloud Spanner, Amazon Aurora, Microsoft Azure Cosmos DB.

Estes são apenas alguns exemplos dos tipos de BD existentes. Cada um possui características e recursos específicos, e a escolha do tipo de BD depende das necessidades do projeto e dos requisitos de armazenamento e acesso aos dados.

2. Criando e manipulando Tabelas no SQL

Para a criação e manupulação de tabelas de dados no SQL, vamos usar a Linguagem SQL. A seguir, um exemplo básico de como criar uma tabela usando o SQLite:

a) Abra um terminal ou console do SQL.
b) Conecte-se ao banco de dados usando o comando sql seguido do nome do arquivo de banco de dados ou o caminho completo para o arquivo. Por exemplo:

   sql meu_banco_de_dados.db

c) Ou fazer a conexão com o banco de dados no link MS SQL, clicar em conexão.

Depois de se conectar ao banco de dados, você pode executar comandos SQL para criar tabelas, **CREATE TABLE** nome_da_tabela. Por exemplo, para criar uma tabela chamada "usuarios" com colunas para "id", "nome" e "idade", você pode usar o seguinte comando:

```
CREATE TABLE usuarios (
id INTEGER PRIMARY KEY,
nome VARCHAR(50),
idade INTEGER
);
```

Neste exemplo, definimos três colunas: "id" como um número inteiro que será a chave primária, "nome" como um varchar de tamanho 50 e "idade" como um número inteiro.

Execute o comando SQL pressionando Enter.

Agora, você tem uma tabela "usuarios" criada no banco de dados SQLite. Você pode inserir dados na tabela usando comandos de inserção (**INSERT**), consultar os dados (**SELECT**), atualizar os dados (**UPDATE**) ou excluir registros (**DELETE**), conforme necessário.

Lembre-se de adaptar o exemplo às suas necessidades, adicionando ou modificando as colunas e tipos de dados conforme necessário. O SQLite oferece uma ampla gama de tipos de dados e recursos SQL para atender às suas necessidades específicas.

Para inserir registros em uma tabela do SQLite, você pode usar o comando SQL **INSERT INTO**. Abaixo um exemplo de como inserir registros na tabela "usuarios" que criamos anteriormente:

```
INSERT INTO usuarios (id, nome, idade) VALUES (1, 'João', 25);
INSERT INTO usuarios (id, nome, idade) VALUES (2, 'Maria', 30);
```

Nesse exemplo, inserimos dois registros na tabela "usuarios". Cada comando INSERT INTO especifica as colunas às quais os valores serão atribuídos. No caso, inserimos os valores 1, 'João' e 25 para as colunas "nome" e "idade" do primeiro registro e 2, 'Maria' e 30 para o segundo registro.

Para mostrar os dados existentes em uma tabela do SQLite, você pode usar o comando SQL **SELECT**. Aqui está um exemplo de como exibir todos os dados da tabela "usuarios":

```
SELECT * FROM usuarios;
```

Nesse exemplo, usamos o asterisco (*) para selecionar todas as colunas da tabela "usuarios". Isso retornará todas as linhas e colunas da tabela.

Se você quiser selecionar colunas específicas, pode listar os nomes das colunas separados por vírgula em vez de usar o asterisco (*). Por exemplo:

SELECT nome, idade FROM usuarios;

Nesse caso, apenas as colunas "nome" e "idade" serão exibidas para cada registro na tabela "usuarios".

Lembre-se de substituir "usuarios" pelo nome da sua tabela real. Depois de executar o comando **SELECT**, o SQLite retornará os dados existentes na tabela de acordo com as colunas e condições especificadas.

Se você quiser mostrar apenas os registros que contêm um valor específico em um campo, você pode usar a cláusula **WHERE** na sua consulta **SELECT**. Aqui está um exemplo de como mostrar os registros da tabela "usuarios" que têm o nome "João":

SELECT * FROM usuarios WHERE nome = 'João';

Nesse exemplo, usamos a cláusula **WHERE** para especificar a condição em que o campo "**nome**" deve ser igual a "**João**", lembrar que aqui a busca é pelo nome “João”, se tiver mais de um nome todos devem ser inseridos, exemplo “João Antonio”. Isso retornará apenas os registros que correspondem a essa condição.

Se você deseja exibir apenas a coluna "**nome**" para os registros com o nome "**João**", você pode ajustar a consulta da seguinte maneira:

SELECT nome FROM usuarios WHERE nome = 'João';

Dessa forma, apenas a coluna "**nome**" será exibida para os registros que atendem à condição.

Lembre-se de adaptar o exemplo ao nome da tabela e ao campo que você está usando no seu banco de dados SQLite. Certifique-se de usar as aspas simples (') ao especificar valores de texto.

3. Ao criar uma Tabela no SQL, você pode definir o tipo de dados e o tamanho dos campos.

Aqui estão alguns exemplos de como especificar o tipo de dados e o tamanho ao criar uma tabela:

Texto/Varchar: Para campos de texto, você pode usar os tipos **TEXT** ou **VARCHAR**. O tamanho máximo para campos **TEXT** é 2^31 bytes, mas você pode especificar um tamanho máximo para campos **VARCHAR**. Por exemplo:

```
CREATE TABLE minha_tabela (
  meu_campo_texto TEXT,
  meu_campo_varchar VARCHAR(50)
);
```

Inteiro: Para campos numéricos inteiros, você pode usar o tipo **INTEGER**. O tamanho do inteiro é de 1, 2, 3, 4, 6 ou 8 bytes, dependendo do valor armazenado. Não há necessidade de especificar o tamanho ao criar um campo **INTEGER**. Exemplo:

```
CREATE TABLE minha_tabela (
  meu_campo_inteiro INTEGER
);
```

Float: Para campos numéricos em ponto flutuante, você pode usar o tipo **REAL** ou **FLOAT**. O tamanho do campo é de 8 bytes. Exemplo:

```
CREATE TABLE minha_tabela (
  meu_campo_float REAL
);
```

Data: Para campos de data, você pode usar o tipo **TEXT** ou **INTEGER**. O SQLite não possui um tipo de dados **DATE** dedicado. Você pode armazenar datas como texto no formato 'YYYY-MM-DD' ou como valores inteiros representando o número de segundos ou milissegundos desde uma data de referência. Exemplo:

```
CREATE TABLE minha_tabela (
  meu_campo_data TEXT,
  meu_campo_data_inteiro INTEGER
);
```

Memo: Se você deseja armazenar textos longos ou blocos de texto, pode usar o tipo TEXT para

```
CREATE TABLE minha_tabela (
  meu_campo_memo TEXT
);
```

Lembre-se de ajustar o nome da tabela, o nome dos campos e os tipos de dados de acordo com suas necessidades. Esses são apenas exemplos de como especificar tipos de dados e tamanhos ao criar uma tabela no SQLite.

Para deletar uma tabela existente no SQLite, você pode usar o comando **SQL DROP TABLE**. Aqui está um exemplo de como deletar uma tabela chamada "minha_tabela":

```
DROP TABLE minha_tabela;
```

Ao executar esse comando, a tabela "minha_tabela" será excluída permanentemente do banco de dados. Certifique-se de ter certeza de que deseja deletar a tabela, pois essa operação não pode ser desfeita.

Lembre-se de substituir "minha_tabela" pelo nome real da tabela que você deseja deletar. Após a execução do comando **DROP TABLE**, todos os dados e esquema relacionados à tabela serão removidos.

4. Criando Tabelas inter-relacionadas por Chaves-Primária e Chave-Secundária ou Chave-Estrangeira.

Aqui está um exemplo de como criar e relacionar duas tabelas no SQLite, uma para alunos e outra para dados de disciplinas e notas, usando chaves primárias e secundárias:

```
-- Tabela Alunos
CREATE TABLE Alunos (
  id_aluno INTEGER PRIMARY KEY,
  nome varchar(50),
  idade INTEGER
);
-- Tabela Disciplinas
CREATE TABLE Disciplinas (
  id_disciplina INTEGER PRIMARY KEY,
  nome varchar(50),
  professor varchar(50)
);
```

```
-- Tabela Notas
CREATE TABLE Notas (
  id_nota INTEGER PRIMARY KEY,
  id_aluno INTEGER,
  id_disciplina INTEGER,
  nota INTEGER,
  FOREIGN KEY (id_aluno) REFERENCES Alunos(id_aluno),
  FOREIGN KEY (id_disciplina) REFERENCES Disciplinas(id_disciplina)
);
```

Nesse exemplo, a **tabela "Alunos"** possui uma **chave primária "id_aluno"** do **tipo INTEGER**, a **tabela "Disciplinas"** possui uma **chave primária "id_disciplina"** do **tipo INTEGER** e a **tabela "Notas"** também possui uma **chave primária "id_nota"** do **tipo INTEGER**.

Além disso, a **tabela "Notas"** possui **duas chaves estrangeiras (id_aluno e id_disciplina) que se referem às chaves primárias das tabelas "Alunos" e "Disciplinas", respectivamente**. **Essas chaves estrangeiras estabelecem uma relação entre as tabelas e permitem a integridade referencial.**

Observe que você pode adicionar mais colunas às tabelas, como desejado, para armazenar informações adicionais sobre os alunos, disciplinas e notas.

Lembre-se de adaptar o exemplo às suas necessidades, adicionando ou modificando as colunas e os tipos de dados conforme necessário.

Para inserir registros nas tabelas "Alunos", "Disciplinas" e "Notas" do exemplo fornecido, você pode usar os comandos INSERT INTO. Aqui está um exemplo de como lançar 3 alunos na tabela "Alunos", 1 disciplina na tabela "Disciplinas" e 3 notas para cada aluno na tabela "Notas":

-- Inserindo alunos na tabela Alunos

```
INSERT INTO Alunos (id_aluno, nome, idade) VALUES (1, 'João', 20);
INSERT INTO Alunos (id_aluno, nome, idade) VALUES (2, 'Maria', 22);
INSERT INTO Alunos (id_aluno, nome, idade) VALUES (3, 'Pedro', 21);
SELECT * FROM Alunos;
```

-- Inserindo disciplina na tabela Disciplinas

```
INSERT INTO Disciplinas (id_disciplina, nome, professor) VALUES (1, 'Matemática', 'Prof. Silva');
SELECT * FROM Disciplinas;
```

-- Inserindo notas para cada aluno na tabela Notas

```
INSERT INTO Notas (id_nota, id_aluno, id_disciplina, nota) VALUES (1,1, 1, 8);
INSERT INTO Notas (id_nota, id_aluno, id_disciplina, nota) VALUES (2,1, 1, 7);
INSERT INTO Notas (id_nota, id_aluno, id_disciplina, nota) VALUES (3,1, 1, 9);

INSERT INTO Notas (id_nota, id_aluno, id_disciplina, nota) VALUES (4,2, 1, 6);
INSERT INTO Notas (id_nota, id_aluno, id_disciplina, nota) VALUES (5,2, 1, 8);
INSERT INTO Notas (id_nota, id_aluno, id_disciplina, nota) VALUES (6,2, 1, 7);

INSERT INTO Notas (id_nota, id_aluno, id_disciplina, nota) VALUES (7,3, 1, 9);
INSERT INTO Notas (id_nota, id_aluno, id_disciplina, nota) VALUES (8,3, 1, 7);
INSERT INTO Notas (id_nota, id_aluno, id_disciplina, nota) VALUES (9, 3, 1, 8);
SELECT * FROM Notas;
```

Nesse exemplo, inserimos três alunos ("João", "Maria" e "Pedro") na tabela "Alunos" com seus respectivos nomes e idades. Em seguida, inserimos uma disciplina ("Matemática" com o professor "Prof. Silva") na tabela "Disciplinas". Por fim, inserimos três notas para cada aluno na tabela "Notas" associando o

aluno pelo "id_aluno", a disciplina pelo "id_disciplina" e inserindo a nota correspondente.

Certifique-se de ajustar os valores dos campos (nomes, idades, nome da disciplina, professor e notas) conforme necessário para seus dados específicos.

Lembre-se de adaptar o exemplo às suas necessidades e garantir que os valores inseridos correspondam às chaves primárias existentes nas tabelas relacionadas ("Alunos" e "Disciplinas") para as chaves estrangeiras da tabela "Notas".

Para alterar um registro já incluído em uma tabela, você pode usar o comando UPDATE no SQLite. Aqui está um exemplo de como alterar um registro em uma tabela específica:

```
-- Alterar o nome de um aluno com o id_aluno igual a 1
UPDATE Alunos SET nome = 'Novo Nome' WHERE id_aluno = 1;
```

Nesse exemplo, usamos o comando UPDATE para alterar o valor do campo nome na tabela Alunos. O operador SET é usado para especificar qual coluna será alterada e qual valor será atribuído a ela. No exemplo acima, estamos alterando o nome do aluno para 'Novo Nome'. A cláusula WHERE é usada para especificar a condição que determina quais registros serão atualizados. No exemplo acima, estamos selecionando o registro com id_aluno igual a 1.

Lembre-se de adaptar o exemplo ao nome da tabela e aos nomes das colunas que você está utilizando. Além disso, você pode adicionar outras condições na cláusula WHERE para refinar a seleção dos registros a serem atualizados.

5. Visualizando as Tabelas interligadas por meu de Relatórios Simples

Para mostrar as notas dos alunos em um formato de relatório, você pode usar uma consulta SELECT com a cláusula JOIN para combinar as informações das

tabelas "Alunos", "Disciplinas" e "Notas". Aqui está um exemplo de como exibir o relatório das notas dos alunos:

```
SELECT Alunos.nome AS NomeAluno, Disciplinas.nome AS NomeDisciplina, Notas.nota AS Nota
FROM Alunos
JOIN Notas ON Alunos.id_aluno = Notas.id_aluno
JOIN Disciplinas ON Notas.id_disciplina = Disciplinas.id_disciplina;
```

Nessa consulta, usamos a cláusula JOIN para combinar as tabelas "Alunos", "Notas" e "Disciplinas" com base nas chaves estrangeiras relacionadas. Em seguida, selecionamos os campos que desejamos exibir no relatório, renomeando-os com aliases (NomeAluno, NomeDisciplina, Nota) para tornar a saída mais legível.

A consulta acima retornará o nome do aluno, o nome da disciplina e a nota correspondente para cada registro na tabela "Notas".

Você pode adaptar o exemplo acima aos nomes reais das suas tabelas e colunas. Além disso, você pode adicionar condições adicionais à consulta, como ordenação ou filtragem, se necessário.

Detalhando o código podemos entender que:

O código apresentado é um exemplo de consulta SQL que combina as tabelas "Alunos", "Notas" e "Disciplinas" usando a cláusula JOIN e retorna o nome do aluno, o nome da disciplina e a nota correspondente.

Aqui está uma explicação linha por linha do código:

```
SELECT Alunos.nome AS NomeAluno, Disciplinas.nome AS NomeDisciplina, Notas.nota AS Nota
```

Nesta linha, estamos selecionando as colunas que queremos exibir no resultado da consulta. Estamos atribuindo aliases (apelidos) para os nomes das colunas usando a palavra-chave AS. Alunos.nome AS NomeAluno significa que queremos exibir o nome do aluno com o alias "NomeAluno", Disciplinas.nome AS NomeDisciplina significa que queremos exibir o nome da disciplina com o alias "NomeDisciplina" e Notas.nota AS Nota significa que queremos exibir a nota com o alias "Nota".

FROM Alunos

JOIN Notas ON Alunos.id_aluno = Notas.id_aluno

JOIN Disciplinas ON Notas.id_disciplina = Disciplinas.id_disciplina;

Nestas linhas, estamos especificando as tabelas envolvidas na consulta e as condições de junção (JOIN). Usamos o JOIN para combinar as tabelas com base nas chaves estrangeiras. Estamos combinando a tabela "Alunos" com a tabela "Notas" usando a condição Alunos.id_aluno = Notas.id_aluno e a tabela "Notas" com a tabela "Disciplinas" usando a condição Notas.id_disciplina = Disciplinas.id_disciplina.

Essa consulta irá retornar os resultados com o nome do aluno, o nome da disciplina e a nota correspondente para cada registro nas tabelas relacionadas.

Lembre-se de adaptar o exemplo aos nomes reais das suas tabelas e colunas. Além disso, você pode adicionar condições adicionais à consulta, como ordenação ou filtragem, se necessário.

6. Criando Tabelas e incrementando automaticamente as chaves primárias

Ao criar tabelas no SQLite, você pode definir a coluna da chave primária com o tipo de dados INTEGER e usar a palavra-chave AUTOINCREMENT para alimentar automaticamente os valores da chave primária. Aqui está um exemplo:

```
-- Tabela Alunos
CREATE TABLE Alunos (
  id_aluno INTEGER PRIMARY KEY AUTOINCREMENT,
  nome VARCHAR(50),
  idade INTEGER
);

-- Tabela Disciplinas
CREATE TABLE Disciplinas (
  id_disciplina INTEGER PRIMARY KEY AUTOINCREMENT,
  nome VARCHAR(50),
  professor VARCHAR(50)
);

-- Tabela Notas
CREATE TABLE Notas (
  id_nota INTEGER PRIMARY KEY AUTOINCREMENT,
  id_aluno INTEGER,
  id_disciplina INTEGER,
  nota INTEGER,
  FOREIGN KEY (id_aluno) REFERENCES Alunos(id_aluno),
  FOREIGN KEY (id_disciplina) REFERENCES Disciplinas(id_disciplina)
);
```

Nesse exemplo, ao definir a coluna da chave primária como **INTEGER PRIMARY KEY AUTOINCREMENT**, o SQLite irá atribuir automaticamente valores

crescentes e únicos para a chave primária ao inserir registros nas tabelas. Isso significa que você não precisa se preocupar em fornecer explicitamente os valores para a chave primária durante a inserção de registros. O SQLite cuidará disso automaticamente.

No entanto, é importante observar que o uso da palavra-chave **AUTOINCREMENT** no SQLite tem algumas implicações de desempenho e uso de armazenamento. Em muitos casos, a coluna **INTEGER PRIMARY KEY** já é suficiente para garantir a unicidade dos valores da chave primária. Portanto, se você não tiver uma necessidade específica de usar **AUTOINCREMENT**, pode simplesmente definir a coluna da chave primária como **INTEGER PRIMARY KEY**.

7. Criando formulário para lançamentos dos dados nas tabelas alunos, disciplinas e notas

Para isso podemos criar um formulário em HTML e para salvar os dados no banco de dados podemos usar a linguagem PHP.

Para criar um formulário para lançamentos de dados para alunos e disciplinas, você precisará usar uma linguagem de programação ou framework web para criar a interface do usuário, como descrito acima. Como exemplo básico vamos utilizar HTML e PHP para criar um formulário simples.

7.1. Crie um arquivo HTML chamado formulario.html com o seguinte conteúdo:

```
<!DOCTYPE html>
<html>
<head>
    <title>Formulário de Lançamento de dados</title>
</head>
<body>
```

```
    <h1>Formulário de Lançamentos</h1>

    <form action="processar_formulario.php" method="POST">
        <h2>Aluno</h2>
        <label for="nome_aluno">Nome do Aluno:</label>
        <input type="text" id="nome_aluno" name="nome_aluno">

        <label for="idade_aluno">Idade do Aluno:</label>
        <input type="number" id="idade_aluno" name="idade_aluno">

        <h2>Disciplina</h2>
        <label for="nome_disciplina">Nome da Disciplina:</label>
        <input type="text" id="nome_disciplina" name="nome_disciplina">

        <label for="professor">Nome do Professor:</label>
        <input type="text" id="professor" name="professor">

        <h2>Notas</h2>
        <label for="nota_aluno">Nota do Aluno:</label>
        <input type="number" id="nota_aluno" name="nota_aluno">

        <br><br>
        <input type="submit" value="Enviar">
    </form>
</body>
</html>
```

7.2. Crie um arquivo PHP para fazer as conexões com as tabelas

Crie,agora, um arquivo PHP chamado processar_formulario.php para processar os dados enviados pelo formulário e realizar as inserções no banco de dados. Certifique-se de que você tem uma conexão válida com o banco de dados SQLite. Aqui está um exemplo de como esse arquivo pode ser:

```
<?php
// Obtém os dados do formulário
$nome_aluno = $_POST['nome_aluno'];
$idade_aluno = $_POST['idade_aluno'];
$nome_disciplina = $_POST['nome_disciplina'];
$professor = $_POST['professor'];
$nota_aluno = $_POST['nota_aluno'];

// Realiza a inserção dos dados no banco de dados
$db = new SQLite3('caminho/para/o/seu/banco-de-dados.db');

// Inserir dados na tabela Alunos
$db->exec("INSERT INTO Alunos (nome, idade) VALUES ('$nome_aluno', $idade_aluno)");

// Obtém o último ID inserido na tabela Alunos
$id_aluno = $db->lastInsertRowID();

// Inserir dados na tabela Disciplinas
$db->exec("INSERT INTO Disciplinas (nome, professor) VALUES ('$nome_disciplina', '$professor')");

// Obtém o último ID inserido na tabela Disciplinas
$id_disciplina = $db->lastInsertRowID();

// Inserir dados na tabela Notas
$db->exec("INSERT INTO Notas (id_aluno, id_disciplina, nota) VALUES ($id_aluno, $id_disciplina, $nota_aluno)");

// Fechar a conexão com o banco de dados
$db->close();

// Redirecionar para uma página de sucesso ou exibir uma mensagem de sucesso
echo "Dados inseridos com sucesso!";
?>
```

Certifique-se de ajustar o código para atender às configurações.

8. Banco de Dados aplicado a manejo de Animais

Para ilustrar melhor a aplicação e uso do Banco de Dados vamos criar um pequeno banco para gerenciar os Animais de uma Fazenda, onde teremos 4 tabelas básicas, que são: Animais, Fazenda, Vacina e Vacinação.

Uma pequena estrutura que tem como objetivo controlar os animais de uma fazenda, bem como as vacinas por eles tomadas.

Então, vamos ao trabalho!!!

Para criar o banco de dados com as tabelas e seus respectivos campos, você pode usar o seguinte código SQL:

```
-- Tabela Animal
CREATE TABLE Animal (
  id_animal INTEGER PRIMARY KEY,
  nome VARCHAR(50),
  mae VARCHAR(50),
  pai VARCHAR(50),
  data_nasc VARCHAR(50),
  data_desm VARCHAR(50),
  peso_nasc REAL,
  peso_desm REAL,
  faz_orig INTEGER,
  FOREIGN KEY (faz_orig) REFERENCES Fazenda(id_faz)
);

-- Tabela Fazenda
CREATE TABLE Fazenda (
```

```
  id_faz INTEGER PRIMARY KEY,
  nome VARCHAR(50),
  proprietario VARCHAR(50),
  telefone VARCHAR(50)
);

-- Tabela Vacina
CREATE TABLE Vacina (
  id_vacina INTEGER PRIMARY KEY,
  nome VARCHAR(50),
  tipo VARCHAR(50),
  data_venc VARCHAR(50),
  fabricante VARCHAR(50)
);

-- Tabela Vacinacao
CREATE TABLE Vacinacao (
  id_vacinacao INTEGER PRIMARY KEY,
  data_vacinacao VARCHAR(50),
  id_animal INTEGER,
  id_vacina INTEGER,
  nome_aplicador VARCHAR(50),
  FOREIGN KEY (id_animal) REFERENCES Animal(id_animal),
  FOREIGN KEY (id_vacina) REFERENCES Vacina(id_vacina)
);
```

Nesse exemplo, criamos quatro tabelas: "Animal", "Fazenda", "Vacina" e "Vacinacao". Cada tabela possui os campos especificados de acordo com a descrição fornecidas pelo usuário (Gestor da Fazenda).

As tabelas "Fazenda" e "Vacina" possuem chaves primárias definidas como "id_faz" e "id_vacina", respectivamente.

A tabela "Animal" possui uma chave estrangeira "faz_orig" que referencia a tabela "Fazenda" pelo campo "id_faz". A tabela "Vacinacao" possui duas chaves estrangeiras: "id_animal" que referencia a tabela "Animal" pelo campo "id_animal" e "id_vacina" que referencia a tabela "Vacina" pelo campo "id_vacina".

Para inserir os registros nas tabelas do banco de dados, você pode executar as instruções de inserção utilizando comandos SQL. Aqui está um exemplo de como inserir os registros nas tabelas com os dados fornecidos, onde vamos inserir 5 animais, 2 fazendas, 3 vacinas e 3 vacinações para cada animal cadastrado.

```
-- Inserir animais na tabela Animal
INSERT INTO Animal (id_animal, nome, mae, pai, data_nasc, data_desm, peso_nasc, peso_desm, faz_orig)
VALUES
  (1, 'Animal 1', 'Mãe 1', 'Pai 1', '2022-01-01', '2022-06-01', 2.5, 200.0, 1),
  (2, 'Animal 2', 'Mãe 2', 'Pai 2', '2022-02-01', '2022-07-01', 3.0, 220.0, 1),
  (3, 'Animal 3', 'Mãe 3', 'Pai 3', '2022-03-01', '2022-08-01', 2.8, 190.0, 2),
  (4, 'Animal 4', 'Mãe 4', 'Pai 4', '2022-04-01', '2022-09-01', 2.2, 180.0, 2),
```

(5, 'Animal 5', 'Mãe 5', 'Pai 5', '2022-05-01', '2022-10-01', 2.6, 210.0, 2);

-- Inserir fazendas na tabela Fazenda

INSERT INTO Fazenda (id_faz, nome, proprietario, telefone)

VALUES

(1, 'Fazenda 1', 'Proprietário 1', '123456789'),

(2, 'Fazenda 2', 'Proprietário 2', '987654321');

-- Inserir vacinas na tabela Vacina

INSERT INTO Vacina (id_vacina, nome, tipo, data_venc, fabricante)

VALUES

(1, 'Vacina 1', 'Tipo 1', '2024-01-01', 'Fabricante 1'),

(2, 'Vacina 2', 'Tipo 2', '2024-02-01', 'Fabricante 2'),

(3, 'Vacina 3', 'Tipo 3', '2024-03-01', 'Fabricante 3');

-- Inserir vacinações na tabela Vacinacao para cada animal cadastrado

INSERT INTO Vacinacao (id_vacinacao, data_vacinacao, id_animal, id_vacina, nome_aplicador)

VALUES

(1, '2022-07-15', 1, 1, 'Aplicador 1'),

(2, '2022-07-20', 1, 2, 'Aplicador 2'),

(3, '2022-08-01', 1, 3, 'Aplicador 3'),

(4, '2022-08-15', 2, 1, 'Aplicador 1'),

(5, '2022-08-20', 2, 2, 'Aplicador 2'),

(6, '2022-09-01', 2, 3, 'Aplicador 3'),

(7, '2022-09-15', 3, 1, 'Aplicador 1'),

(8, '2022-09-20', 3, 2, 'Aplicador 2'),

(9, '2022-10-01', 3, 3, 'Aplicador 3'),

(10, '2022-10-15', 4, 1, 'Aplicador 1'),

(11, '2022-10-20', 4, 2, 'Aplicador 2'),

(12, '2022-11-01', 4, 3, 'Aplicador 3'),

(13, '2022-11-15', 5, 1, 'Aplicador 1'),

(14, '2022-11-20', 5, 2, 'Aplicador 2'),

(15, '2022-12-01', 5, 3, 'Aplicador 3');

Para lançar mais de um registro em uma mesma tabela devemos proceder como acima, onde os valores para cada registro devem ser separados por vírgula (,) e ao finalizar as inserções para a tabela ativa devemos finalizar com um ponto de vírgula (;).

8.1. Gerando Relatórios básicos das Tabelas

Para gerar relatórios a partir das tabelas do banco de dados, você pode utilizar consultas SQL e formatar os resultados de acordo com suas necessidades. Abaixo estão exemplos de consultas para cada tabela:

Relatório da Tabela Animal:

```
SELECT id_animal, nome, mae, pai, data_nasc, data_desm, peso_nasc, peso_desm, faz_orig FROM Animal;
```

Relatório da Tabela Fazenda:

```
SELECT id_faz, nome, proprietario, telefone FROM Fazenda;
```

Relatório da Tabela Vacina:

SELECT id_vacina, nome, tipo, data_venc, fabricante FROM Vacina;

Relatório da Tabela Vacinacao:

SELECT id_vacinacao, data_vacinacao, id_animal, id_vacina, nome_aplicador FROM Vacinacao;

Essas consultas retornarão todos os registros de cada tabela. Você pode adicionar cláusulas WHERE e ORDER BY para filtrar e ordenar os resultados, se necessário. Lembre-se de substituir os nomes das tabelas e colunas pelos corretos, de acordo com sua estrutura de banco de dados.

Após executar as consultas, você pode formatar e exibir os resultados conforme a preferência, seja em uma tabela HTML, em um arquivo CSV ou qualquer outro formato desejado.

8.2. Ordenando os dados em um relatório

Vamos a um exemplo de consulta utilizando o comando ORDER BY para ordenar os resultados da tabela "Animal" pelo campo "nome" em ordem crescente:

SELECT id_animal, nome, mae, pai, data_nasc, data_desm, peso_nasc, peso_desm, faz_orig FROM Animal ORDER BY nome ASC;

Nesse exemplo, utilizamos a cláusula ORDER BY seguida pelo nome da coluna pelo qual queremos ordenar os resultados. A opção ASC indica que a ordenação deve ser feita em ordem crescente (do menor para o maior valor). Se você quiser

ordenar em ordem decrescente (do maior para o menor valor), você pode utilizar a opção DESC em vez de ASC.

Você também pode adicionar mais campos na cláusula ORDER BY para definir uma ordenação secundária. Por exemplo, se quiser ordenar primeiro pelo nome e, em seguida, pelo peso de nascimento, você pode fazer o seguinte:

```
SELECT id_animal, nome, mae, pai, data_nasc, data_desm, peso_nasc, peso_desm, faz_orig
FROM Animal
ORDER BY nome ASC, peso_nasc DESC;
```

Isso irá ordenar os resultados pelo campo "nome" em ordem crescente e, em caso de empate, pelo campo "peso_nasc" em ordem decrescente.

Lembre-se de ajustar o nome da tabela e os nomes das colunas de acordo com a sua estrutura de banco de dados.

Como observado acima pode-se escrever o comando em uma só linha, ou podemos usar uma linha para cada comando. Entretanto, uma ressalva aqui, quando é usado um comando para cada linha não se deve colocar ao final o ponto e vírgula (;), pois o programa entenderá como fim de linha, fim do comando e, não retornará o que foi solicitado, em função de erro na construção do comando.

9. Outros comandos básicos do SQL

Existem vários outros comandos e recursos que podem ser úteis ao trabalhar com SQL. Aqui estão alguns exemplos adicionais de comandos e recursos que você pode explorar:

1. UPDATE: Utilizado para modificar os dados existentes em uma tabela.
2. DELETE: Utilizado para remover registros de uma tabela.
3. SELECT DISTINCT: Utilizado para retornar valores distintos em uma coluna de uma consulta.
4. GROUP BY: Utilizado para agrupar registros com base em determinadas colunas e aplicar funções de agregação, como SUM, COUNT, AVG, entre outras.
5. HAVING: Utilizado para filtrar resultados após a aplicação do GROUP BY.
6. JOIN: Utilizado para combinar registros de duas ou mais tabelas com base em uma condição relacionada.
7. INDEX: Utilizado para criar índices em colunas específicas para melhorar o desempenho de consultas.
8. ALTER TABLE: Utilizado para modificar a estrutura de uma tabela existente, adicionando, modificando ou excluindo colunas.
9. VIEW: Utilizado para criar uma visualização virtual de uma ou mais tabelas, permitindo consultas simplificadas e reutilizáveis.
10. SUBQUERIES: Utilizado para realizar consultas aninhadas, onde uma consulta é usada como parte de outra consulta.
11. TRIGGER: Utilizado para definir ações automáticas que são executadas em resposta a eventos específicos, como inserção, atualização ou exclusão de registros.
12. TRANSACTIONS: Utilizado para agrupar um conjunto de operações em uma unidade lógica e garantir a atomicidade, consistência, isolamento e durabilidade dos dados.

Esses são apenas alguns exemplos, e existem muitos outros recursos e comandos disponíveis em SQL. É recomendável explorar a documentação do sistema de gerenciamento de banco de dados específico que você está utilizando para obter mais informações detalhadas sobre os comandos e recursos disponíveis.

A seguir estrutura de cada um dos comandos mencionados acima:

1. UPDATE:

```
UPDATE tabela
SET coluna = novo_valor
WHERE condição;
```

2. DELETE:

```
DELETE FROM tabela
WHERE condição;
```

3. SELECT DISTINCT:

```
SELECT DISTINCT coluna
FROM tabela;
```

4. GROUP BY:

```
SELECT coluna, função_agregação(outras_colunas)
FROM tabela
GROUP BY coluna;
```

5. HAVING:

```
SELECT coluna, função_agregação(outras_colunas)
FROM tabela
GROUP BY coluna
HAVING condição;
```

6. JOIN:

```
SELECT colunas
FROM tabela1
JOIN tabela2 ON tabela1.coluna = tabela2.coluna;
```

7. INDEX:

```
CREATE INDEX nome_indice
ON tabela (coluna);
```

8. ALTER TABLE:

```
ALTER TABLE tabela
ADD nova_coluna tipo_dados;

ALTER TABLE tabela
MODIFY coluna tipo_dados;

ALTER TABLE tabela
DROP COLUMN coluna;
```

9. VIEW:

```
CREATE VIEW nome_view AS
SELECT colunas
FROM tabela
WHERE condição;
```

10. SUBQUERIES:

```
SELECT colunas
FROM tabela
WHERE coluna IN (SELECT coluna FROM outra_tabela);

SELECT colunas
FROM tabela
WHERE coluna = (SELECT coluna FROM outra_tabela);

SELECT colunas
FROM tabela
WHERE coluna = (SELECT MAX(coluna) FROM outra_tabela);
```

11. TRIGGER:

```
CREATE TRIGGER nome_trigger
AFTER INSERT ON tabela
FOR EACH ROW
BEGIN
-- ações a serem executadas
END;
```

12. TRANSACTIONS:

```
BEGIN TRANSACTION;
-- operações de inserção, atualização ou exclusão

COMMIT; -- confirma as operações
ROLLBACK; -- desfaz as operações
```

Lembre-se de ajustar os nomes das tabelas, colunas e demais elementos de acordo com a sua estrutura de banco de dados que você implementou. Além disso, alguns comandos podem ter variações de sintaxe dependendo do sistema de gerenciamento de banco de dados (SGBD) que você esteja utilizando. Normalmente, estas sintaxes funcionam na maioria dos SGDB.

Para facilitar ainda mais o aprendizado vamos dar exemplos usando as Tabelas Animal e Fazenda.

1. SELECT: Exemplo de consulta para recuperar todos os animais cadastrados na tabela "Animal":

```
SELECT id_animal, nome, mae, pai, data_nasc, peso_nasc
FROM Animal;
```

2. INSERT: Exemplo de comando para inserir um novo animal na tabela "Animal":

```
INSERT INTO Animal (nome, mae, pai, data_nasc, peso_nasc,
faz_orig)
```

VALUES ('Fiona', 'Molly', 'Buddy', '2022-01-15', 4.5, 'Fazenda XYZ');

3. UPDATE: Exemplo de comando para atualizar o peso de nascimento de um animal específico na tabela "Animal":

   UPDATE Animal

   SET peso_nasc = 5.2

   WHERE id_animal = 1;

4. DELETE: Exemplo de comando para excluir um animal da tabela "Animal":

   DELETE FROM Animal

   WHERE id_animal = 3;

5. JOIN: Exemplo de consulta para obter os detalhes de um animal e sua fazenda de origem usando a tabela "Animal" e "Fazenda":

   SELECT Animal.nome AS NomeAnimal, Animal.data_nasc, Fazenda.nome AS NomeFazenda, Fazenda.proprietario

   FROM Animal

   JOIN Fazenda ON Animal.faz_orig = Fazenda.id_faz

   WHERE Animal.id_animal = 1;

6. SELECT DISTINCT: Exemplo de consulta para obter os nomes únicos dos animais cadastrados na tabela "Animal":

   SELECT DISTINCT nome

   FROM Animal;

7. GROUP BY: Exemplo de consulta para obter a contagem de animais agrupados por fazenda de origem na tabela "Animal":

   SELECT faz_orig, COUNT(*) AS TotalAnimais

   FROM Animal

   GROUP BY faz_orig;

8. HAVING: Exemplo de consulta para obter fazendas que possuem mais de 3 animais na tabela "Animal":

```
SELECT faz_orig, COUNT(*) AS TotalAnimais
FROM Animal
GROUP BY faz_orig
HAVING COUNT(*) > 3;
```

9. INDEX: Exemplo de comando para criar um índice na coluna "nome" da tabela "Animal":

```
CREATE INDEX idx_nome_animal
ON Animal (nome);
```

10. ALTER TABLE: Exemplo de comando para adicionar uma nova coluna chamada "data_vacina" na tabela "Vacinação":

```
ALTER TABLE Vacinação
ADD COLUMN data_vacina DATE;
```

Exemplos: Alteração de nome de campo, ou Inserção de campo na tabela Alunos:

1. **ALTER TABLE Alunos RENAME COLUMN nome TO nome_aluno;**
2. **ALTER TABLE Alunos ADD COLUMN mae varchar(50);**

Estes são apenas alguns exemplos adicionais para explorar os comandos mencionados anteriormente. Lembre-se de adaptar os exemplos às suas tabelas e colunas específicas para evitar qualquer problema com os nomes usados como exemplos.