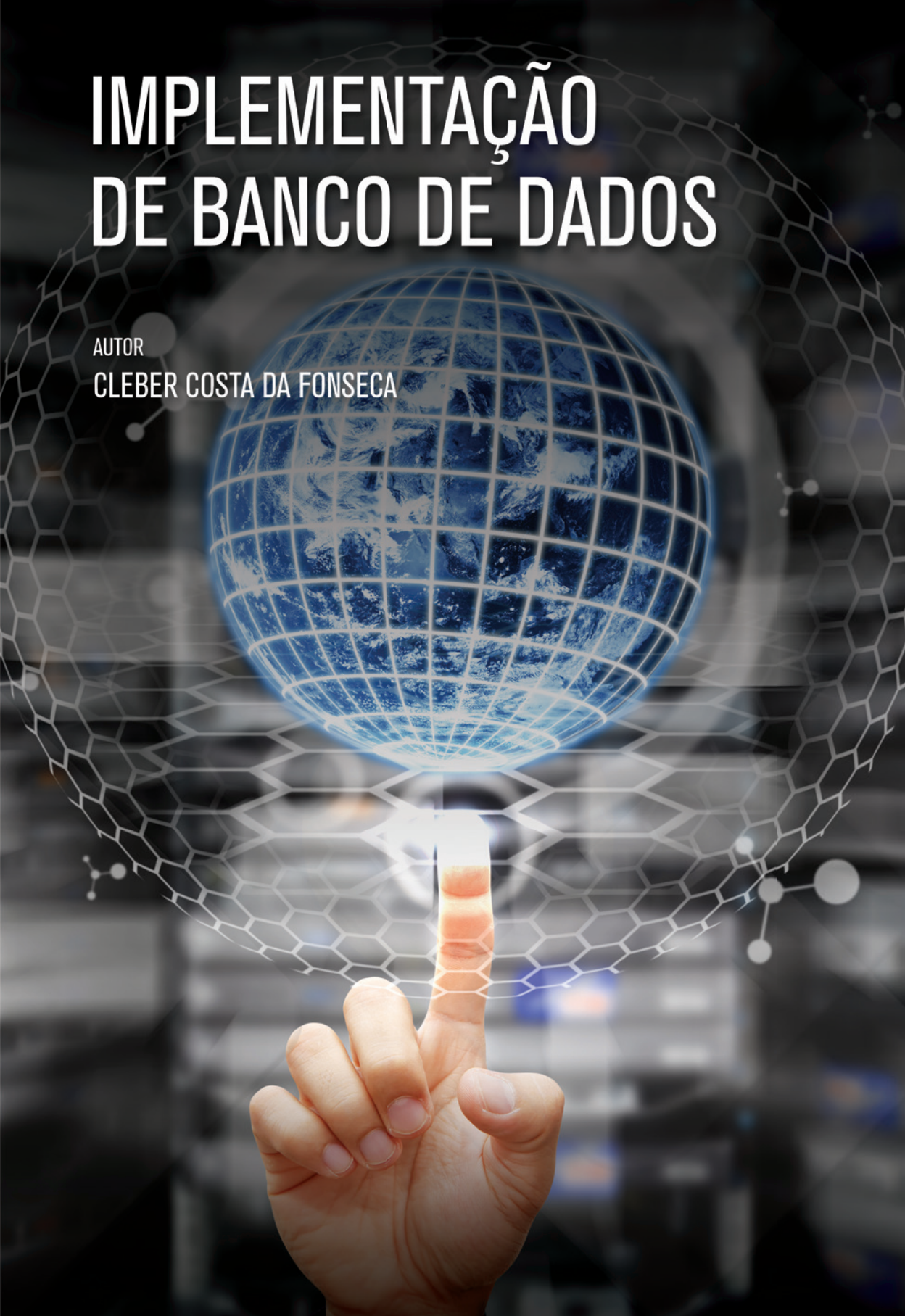
IMPLEMENTAÇÃO DE BANCO DE DADOS

AUTOR
CLEBER COSTA DA FONSECA



IMPLEMENTAÇÃO DE BANCO DE DADOS

AUTOR
CLEBER COSTA DA FONSECA

1ª EDIÇÃO
SESES
RIO DE JANEIRO 2016

Estácio

Conselho editorial REGIANE BURGER, ROBERTO PAES E PAOLA GIL DE ALMEIDA

Autor do original CLEBER COSTA DA FONSECA

Projeto editorial ROBERTO PAES

Coordenação de produção PAOLA GIL DE ALMEIDA

Projeto gráfico PAULO VITOR BASTOS

Diagramação BFS MEDIA

Revisão linguística BFS MEDIA

Revisão de conteúdo SIDNEY NICOLAU VENTURI FILHO

Imagem de capa WATCHARAKUN | SHUTTERSTOCK.COM

Dados Internacionais de Catalogação na Publicação (CIP)

F676i FONSECA, CLEBER COSTA DA

Implementação de banco de dados / Cleber Costa da Fonseca.

Rio de Janeiro: SESES, 2016.

112 P: IL.

ISBN: 978-85-5548-297-7

1. LÓGICA. 2. RELACIONAL. 3. BANCO DE DADOS. 4. ORACLE. 5. SQL.

I. SESES. II. ESTÁCIO.

CDD 005.74

Diretoria de Ensino — Fábrica de Conhecimento

Rua do Bispo, 83, bloco F, Campus João Uchôa

Rio Comprido — Rio de Janeiro — RJ — CEP 20261-063

Sumário

Prefácio

Prezados(as) alunos(as),

Bem-vindos ao curso de Implementação de Banco de Dados. Nesta disciplina veremos as principais teorias empregadas aos Bancos de Dados Relacionais. Serão abordadas técnicas e métodos referentes ao modelo relacional de dados bem como apresentadas as principais ferramentas para o desenvolvimento, confecção, manutenção e organização de um banco de dados relacional.

Vocês, alunos e alunas, terão com esse livro, a condição de experimentar e aprender a teoria envolvida em um SGBD (Sistemas Gerenciadores de Banco de Dados Relacionais). Aprenderão a criar, alimentar, organizar e manter um banco de dados relacional e, com isso, facilitando, assim, a extração de informações desse banco.

Serão cinco capítulos que ajudarão a familiarizarem-se com a linguagem SQL, onde aprenderão a criar suas tabelas de dados, entender como são feitos os relacionamentos entre essas tabelas e como disponibilizar esses dados relacionados por meio de consultas SQL. Irão descobrir como melhorar essas consultas por meio de criação de índices e análises das estruturas. Irão descobrir como o modelo Relacional garante a integridade dos dados e como são feitas transações sem a perda ou incoerência de dados.

A informação na tomada de uma decisão é primordial. A informação organizada e bem estruturada facilita e direciona à melhor decisão.

Dados, Metadados, Informação e Conhecimento

Para entendermos como funciona um banco de dados relacional vamos voltar um pouco ao início da organização dos dados. E antes ainda, vamos definir o significado de dados, metadados, informação e conhecimento, para então dizer como foi definido o modelo relacional de dados.

Olhando rapidamente, as três palavras poderiam muito bem serem sinônimos. E para algumas áreas de estudo, realmente o são. A temperatura medida em um determinado momento do dia pode muito bem ser um “dado”, por exemplo, 25°C, indicando assim a “informação” de clima agradável no momento, que também pode ser o “conhecimento” daquele momento. Mas olhando de um prisma mais amplo, poderíamos querer “classificar” melhor esses dados para um “conhecimento” mais significativo da situação.

Ao medirmos a temperatura de uma determinada localidade, e usando metodologias mais rígidas (medição da temperatura em uma área controlada e tendo como referência exatamente um determinado momento) teremos um conjunto de dados para podermos analisar melhor e obter um conhecimento inerente a esses dados classificados. Sendo assim, podemos definir melhor:

DADOS: de uma maneira mais genérica, pode-se iniciar o estudo desse termo apresentando a definição de **FERREIRA** et al. (1999, p. 62) ***dado*** é o "princípio em que se assenta uma discussão" ou o "elemento ou base para a formação de um juízo". Ainda, tomando-se um ponto de vista mais filosófico, ***dado*** é "o que se apresenta à consciência como imediato, não construído ou não elaborado", **FERREIRA** et al. (1999, p. 602). Essas definições são úteis para exemplificar o termo dado e situar sua definição de um ponto de vista mais humano.

Computacionalmente falando, pode-se definir dado como um valor armazenado e que por si só não quer dizer muita coisa. Uma lista de números, por exemplo, 10, 12, 20, 21, 23, 38 não nos fornecem nenhum significado, mas por assim dizer, são os dados obtidos e anotados de alguma forma. A partir do momento que se rotula a lista, por exemplo, IDADE: 10, 12, 20, 21, 23, 38, tem-se a definição de METADADOS, ou seja, é o dado explicado através de um nome.

Seguindo o raciocínio, tem-se o termo informação, que quer dizer o dado processado, isto é, o resultado do processamento dos dados 10, 12, 20, 21, 23, 38 respondendo a alguma pergunta. Por exemplo, quais as IDADES acima de 18 anos armazenadas? Após PROCESSAR os dados temos 4 idades, ou seja, 20, 21, 23, 38 anos.

Por fim, chega-se ao conhecimento, que nada mais é o que se deseja obter com a informação. No exemplo sabe-se que são 4 idades das 6 armazenadas acima de 18 anos. Após "processar" os dados com a pergunta, "quantas pessoas são maiores de 18 anos" temos o "conhecimento" de 4 idades.

Resumindo:

- **Dados:** valor armazenado;
- **Metadado:** identificação do dado;
- **Informação:** Processamento dos dados mediante uma pergunta;
- **Conhecimento:** Deduções obtidas a partir das informações.

Bons estudos!

1

Modelo Relacional

1. Modelo Relacional

Após o conhecimento básico de dados e seus desdobramentos, pode-se evoluir mais um pouco e imaginar como toda essa informação pode ser armazenada e recuperada para se obter o conhecimento.

No princípio, essas listas de valores eram armazenadas de forma despreocupada. Apenas sabia-se da existência da lista de idades, lista de temperaturas médias obtidas em determinada cidade, lista dos nomes das pessoas de uma cidade ou qualquer outra lista guardada em um sistema de maneira desproposital.

À medida que essas listas foram ficando complexas e volumosas, a obtenção da informação e conhecimento foram ficando cada vez mais complexos e custosos, demandando tempo de processamento e obtendo-se informações de pouca qualidade.

Com o objetivo de melhor classificar esses dados e consequentemente obter a informação o mais rápido e fidedigna possível, alguns modelos matemáticos foram aplicados a esse conjunto de dados. Um deles foi o modelo com base na Teoria Matemática dos Conjuntos e na Álgebra Relacional, chegando-se ao Modelo de Dados Relacional. Esse modelo é uma ferramenta de modelação de dados com o objetivo principal de, como o próprio nome diz, relacionar os dados entre si.

O Modelo de dados Relacional tem as seguintes vantagens:

- É independente das linguagens de programação;
- É independente dos sistemas de gestão de bases de dados.

OBJETIVOS

- Conhecer os conceitos de Dominio, Relação, Variáveis e Valores no universo do Modelo Relacional;
- Definir as regras de integridade relacional;
- Conhecer e aplicar a Álgebra Relacional.

1.1 Introdução ao Modelo Relacional

"O modelo relacional representa o banco de dados como uma coleção de relações". (ELMASRI, R.; NAVATHE, S., Sistemas de Banco de Dados, pag. 90). Pode-se imaginar uma coleção (e de fato é representado) por uma tabela de valores onde cada linha da tabela representa uma coleção de dados ou valores relacionados. Cada linha da tabela representa uma realidade ligada ao mundo real. O nome da tabela e o nome das colunas são definidos de forma que representem essa realidade. Pode-se ter uma tabela chamada ALUNO onde cada linha venha a ser o Nome de um Aluno a ser armazenado. Podemos melhorar ainda mais essa representação, apresentando uma tabela com as colunas, NOME, NÚMERO DE MATRICULA e CLASSE. Cada linha da tabela representa várias informações de um ALUNO e cada coluna em isolado, representa uma informação específica desse aluno. Assim, o nome da tabela e os nomes das colunas é capaz de dizer o que cada linha representa e também, o que o conjunto representa.

No modelo relacional formal, cada linha é chamada de *tupla,* o nome da coluna é conhecido como *atributo* ou *variável*, e a tabela, *relação*. O tipo de dados que representa os valores que possam existir em cada coluna é o *domínio*.

1.1.1 Domínios, Relações, Variáveis e Valores

Matematicamente, Domínio de uma função são todos os valores possíveis de resposta que satisfazem essa função. Imaginando-se que nossa função seja os números de conta bancária, o domínio da função conta bancária é o conjunto de todos os números de conta. Sendo, conta a variável ou atributo em questão. Trazendo essa informação para uma agência bancária. O domínio da função conta bancária são todos os números de conta daquela agência específica. Para a representação englobando toda uma instituição bancária com todas as suas agências, pode-se definir um domínio Agência, representando o conjunto de todos os nomes de agência daquela instituição financeira. Um **tipo de dado** ou formato deve ser especificado para cada domínio. No exemplo acima, o **tipo de dado** para o domínio Agência, pode ser definido como 9999 ou seja, quatro números.

EXEMPLO

Dom(V_1): domínio de conta (atributo ou variável conta); conjunto de todos os números de conta.
Dom(V_n): domínio de agência (atributo ou variável agência); conjunto de todos os números de agência.

Um esquema de Relação **Rel**, indicada por Rel(V_1, V_2, ..., V_n), é composta por um nome de relação R e de um conjunto de atributos ou variáveis V_1, V_2, ..., V_n. Cada valor V é o nome de um papel desempenhado por um domínio **Dom.** Os índices das variáveis são também ditos, **graus de relação**. Para o exemplo de agência bancária e conta bancária, tem-se uma relação de grau 2 para o esquema de relação Banco.

Uma relação (ou estado da relação) r(R) é uma relação matemática de grau n nos domínios dom(V_1), dom(V_2),..., dom(V_n), que é um subconjunto do produto cartesiano dos domínios que definem R: r(R) Q (dom(V_1) x dom(V_2) x ... x dom(V_n)). O produto cartesiano especifica todas as possíveis combinações de valores dos domínios subjacentes. Então, se indicamos o número total de valores, ou cardinalidade, em um domínio D por I D I (presumindo que todos os domínios sejam finitos), o número total de tuplas no produto cartesiano é | dom(V_1) | x I dom(V_2) | x ... x I dom(V_n) | De todas essas possíveis combinações, um estado de relação em um dado momento — estado de relação corrente — reflete apenas as tuplas válidas que representam um estado em particular do mundo real.

1.1.2 Regras de Integridade Relacional

As regras de Integridade Relacional visam garantir a fidelidade de informações em um banco de dados. Basicamente são três as formas mais comuns:

- **Integridade de Domínio**: diz respeito aos dados que são permitidos nas colunas da relação (tabela);
- **Integridade de Entidade**: diz respeito a unicidade de linhas da relação;
- **Integridade Referencial**: diz respeito a consistência entre as tuplas de relações.

As regras de integridade também são conhecidas como restrições (em inglês, *constraints*).

As restrições são as normas aplicadas em colunas de dados na tabela. Estes são utilizados para limitar o tipo de dados que pode entrar em uma tabela ou relação. Isso garante a precisão e confiabilidade dos dados no banco de dados.

Restrições podem ser definidas a nível de coluna ou tabela. Restrições a nível de coluna são aplicadas somente a uma coluna, enquanto que restrições a nível de tabela são aplicadas a toda a tabela.

A seguir algumas restrições disponíveis no SQL mais usadas.

- **NOT NULL**: Garante que a coluna não pode ter um valor NULL;
- **DEFAULT**: Fornece um valor padrão para uma coluna quando nenhum é especificado;
- **UNIQUE**: Garante que todos os valores em uma coluna são diferentes;
- **PRIMARY KEY**: Identifica cada linha / registro, de maneira única, em uma tabela de banco de dados;
- **FOREIGN KEY**: Identifica linhas / registros relacionada a uma outra tabela;
- **CHECK**: A restrição CHECK garante que todos os valores em uma coluna satisfaçam determinadas condições;
- **INDEX**: Usada para criar e recuperar dados mais rapidamente.

As restrições podem ser especificadas durante a criação de uma tabela ou inserida / modificada a medida que necessidades são encontradas.

Geralmente, as restrições são divididas em três categorias principais:

- **Restrições baseadas em Modelo**: Restrições inerentes ao modelo construído;
- **Restrições baseadas em esquema**: Restrições definidas na criação do modelo, normalmente especificações em DDL (*data definition language* – linguagem de definição de dados);
- **Restrições baseadas em aplicação**: Restrições que não podem ser expressas diretamente no modelo construído, por isso, são deixadas a cargo da aplicação ou programa que fará o uso do modelo de dados.

1.1.2.1 Restrições em Domínio

As Restrições de Domínio, especificam que, dentro de cada tupla, o valor de cada atributo V deve ser um valor atômico do domínio. Por atômico entende-se que cada valor do domínio é indivisível no que diz respeito ao modelo relacional.

Restrições de Domínio são as formas mais básicas de restrições usadas na integridade relacional. Eles são fáceis de testar para quando os dados são inseridos.

Os tipos de dados associados aos domínios incluem os tipos de dados numéricos padrões existentes (inteiro, reais, caracteres, booleanos etc.).

EXEMPLO

Considere os seguintes atributos:

- NOME_PAI
- NOME_ALUNO
- DISCIPLINA
- NOTA

É razoável deduzir que NOME_PAI e NOME_ALUNO fazem parte de um mesmo domínio. Percebe-se que para a tupla NOME_PAI, NOME_ALUNO é garantido a atomicidade de dados, definindo-se o tipo NOME_PESSOA.

É razoável deduzir que DISCIPLINA e NOTA são partes de domínios diferentes.

1.1.2.2 Integridade Referencial

As regras de Integridade Referencial asseguram que se uma tupla faz referência a outra relação deve existir uma tupla válida para essa relação.

Para definir Integridade Referencial, deve-se apresentar o conceito de *chave estrangeira* (FK). Que define uma regra de integridade referencial entre duas relações. Informalmente, uma *chave estrangeira* (FK) é uma coluna ou combinação de colunas que é usada para estabelecer e impor uma relação entre os dados em duas tabelas. Você pode criar uma chave estrangeira definindo uma restrição FOREIGN KEY ao criar ou modificar uma tabela. Nessa relação a coluna da primeira tabela é chamada **relação referência** e a coluna da segunda tabela, chamada de **relação referida**.

EXEMPLO

Considere os seguintes atributos:

- NOME_ALUNO
- NOME_DISCIPLINA
- NOTA

Ao criar uma tabela para "lançar" as notas de uma disciplina de determinados alunos em uma tabela DISCIPLINA_NOTA vemos que a Coluna 'NOME_DISCIPLINA', pode aparecer várias vezes com o mesmo nome. Pode-se então, criar uma outra tabela chamada 'DISCIPLINA' e fazer uma referência a tabela DISCIPLINA_NOTA:

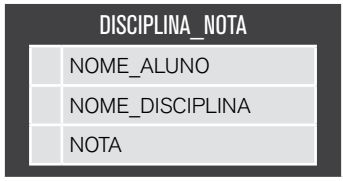


Figura 1.1 – DISCIPLINA_NOTA sem relacionamento. Fonte: Elaborada pelo autor..

Figura 1.2 – DISCIPLINA_NOTA com relacionamento. Fonte: Elaborada pelo autor.

1.1.3 Álgebra Relacional

É o conjunto básico de operações para o modelo relacional. Essas operações permitem a recuperação de tuplas mediante instruções de consulta aplicadas ao banco de dados. O resultado dessa recuperação também será uma relação, que pode ser usada em outras operações de consulta.

A importância da álgebra relacional:

- Provê fundamento formal para operações do modelo relacional;
- Alguns de seus conceitos são incorporados na linguagem SQL padrão.
- E o mais importante, é usada como base para implementar e otimizar as consultas em sistemas de banco de dados relacional;

Pode-se considerar a álgebra relacional como parte do modelo relacional de dados dividindo-se suas operações em dois grupos principais. Um grupo derivado da teoria de conjuntos matemático, incluindo UNION (UNIÃO), INTERSECTION (INTERSEÇÃO), SET DIFFERENCE (DIFERENÇA ENTRE CONJUNTOS) e CROSS PRODUCT (PRODUTO CARTESIANO). O outro grupo, com instruções específicas para os bancos de dados relacionais, incluindo SELECT (SELEÇÃO), PROJECT (PROJEÇÃO) e JOIN (JUNÇÃO).

1.1.3.1 Operações de Seleção e Projeção

As operações de SELECT (SELEÇÃO) E PROJECT (PROJECAO) são ditas **operações unárias,** pois atuam em relações únicas (ELMASRI, R.; NAVATHE, S., Sistemas de Banco de Dados, pag. 106).

A operação SELECT recupera os dados de uma ou mais tabelas, selecionando um subconjunto de tuplas que satisfaça determinada **condição de seleção**. Essa condição de seleção é usada para dividir horizontalmente uma relação em dois conjuntos de tuplas – as tuplas que satisfazem a condição e são retornadas e as tuplas que não satisfazem a condição e são ignoradas.

Em geral, a operação SELEÇÃO é indicada por

σ<condição de seleção>(R)

Em que o símbolo σ (sigma) é usado para indicar o operador SELEÇÃO e a condição de seleção é uma expressão booleana, especificada nos atributos da relação R. Observe que R é, geralmente, uma expressão de álgebra relacional cujo resultado é uma relação — a mais simples delas seria exatamente o nome de uma relação do banco de dados. A relação resultante da operação SELEÇÃO tem os mesmos atributos que R.

Na "condição de seleção" podem ser inseridos os operadores lógicos AND, OR e NOT, definido pelos símbolos: $\wedge$ (AND), $\vee$ (OR) e $\neg$ (NOT).

EXEMPLO

	NOME_ALUNO	NOME_DISCIPLINA	NOTA
1	José Geraldo	Álgebra	8
2	Eduardo Tomaz	Álgebra	10
3	Cleber Dutra	Álgebra	9
4	Hernesto Paula	Física	10
5	Josué José	Álgebra	10

Figura 1.3 – Tabela DISCIPLINA_NOTA. Fonte: Elaborada pelo autor.

Para selecionar os alunos da disciplina 'álgebra' cuja nota foi integral, pode-se especificar cada uma dessas condições em uma operação de SELEÇÃO:

NOME_DISCIPLINA = 'ALGEBRA' AND NOTA = 10

Uma instrução SQL padrão, teremos:

SELECT * FROM DISCIPLINA_NOTA WHERE NOME_DISCIPLINA = 'Álgebra' AND NOTA = 10

A álgebra relacional ficaria:

$\sigma_{NOME\_DISCIPLINA = 'Álgebra' \wedge NOTA = 10}(DISCIPLINA\_NOTA)$

Resultando a nova relação:

	NOME_ALUNO	NOME_DISCIPLINA	NOTA
1	Eduardo Tomaz	Álgebra	10
2	Josué José	Álgebra	10

Figura 1.4 – Resultado de uma operação SELECT. Fonte: Elaborada pelo autor.

A operação PROJECT recupera os dados de certas colunas de uma tabela e descarta outras. Se existir a necessidade de mostrar apenas alguns atributos de uma tabela em detrimento a outros, usa-se a operação PROJECT.

Em geral, a operação PROJECT é de dada por

Π_{Π} <lista de atributo>(R).

Se em uma recuperação PROJECT a lista de atributos forem atributos de R que não são chave, é possível que ocorram tuplas repetidas. A operação PROJECT remove quaisquer tuplas repetidas, garantindo o resultado como um conjunto de tuplas válidas.

EXEMPLO

Para selecionar as disciplinas e as ocorrências de notas nas disciplinas, teremos:

```
SELECT DISTINCT NOME_DISCIPLINA, NOTA FROM DISCIPLINA_NOTA
```

A álgebra relacional ficaria:

$\varpi_{\varpi NOME\_DISCIPLINA\ \varpi NOTA}$ (DISCIPLINA_NOTA)

Resultando a relação:

	NOME_DISCIPLINA	NOTA
1	Álgebra	8
2	Álgebra	10
3	Álgebra	9
4	Física	10

Figura 1.5 – Resultado de uma operação PROJECT. Fonte: Elaborada pelo autor.

1.1.3.2 Operações de Conjuntos

São operações derivadas das operações matemáticas padrão definidas a partir da teoria dos conjuntos. São elas:

- UNION (UNIÃO);
- INTERSECTION (INTERSEÇÃO);
- MINUS (SUBTRAÇÃO).

UNION é a operação de UNIAO da teoria de conjuntos. Se temos as relações $R(A_1, A_2, ..., A_n)$ e $S(B_1, B_2, ..., B_n)$ para haver a operação União, os atributos de cada relação devem ser compatíveis entre si, ou seja, devem ter o mesmo grau (n) e os domínios de cada atributo deve ser igual ao domínio do outro atributo, ou seja, $dom(A_1) = dom(B_1)$. Significando que cada relação possui o mesmo grau e que cada par de atributos possuem o mesmo domínio. Sendo assim a operação UNION pode ser aplicada. O resultado da união, como na representação matemática, indicado por R U S.

EXEMPLO

Temos a tabela ALUNO e a nova tabela PROFESSOR. Podemos fazer a união dos domínios 'NOME_PESSOA' com os nomes dos alunos e os nomes dos professores.

Com os seguintes domínios:

	NOME_PESSOA
1	Cleber Dutra
2	Elmasri
3	Fagundes Teles
4	Ferreira
5	Navathe

Figura 1.6 – Tabela PROFESSOR. Fonte: Elaborada pelo autor.

	NOME_PESSOA
1	Cleber Dutra
2	Eduardo Tomaz
3	Fagundes Teles
4	Hernesto Paula
5	José Geraldo
6	Josué José

Figura 1.7 – Tabela ALUNO. Fonte: Elaborada pelo autor.

A álgebra relacional ficaria:

NOME_PESSOA(PROFESSOR) U NOME_PESSOA(ALUNO)

Resultado da operação UNION:

	NOME_PESSOA
1	Cleber Dutra
2	Eduardo Tomaz
3	Elmasri
4	Fagundes Teles
5	Ferreira
6	Hernesto Paula
7	José Geraldo
8	Josué José
9	Navathe

Figura 1.8 – Operação UNION. Fonte: Elaborada pelo autor.

Da mesma forma como foi apresentado a operação UNION, pode-se trazer a definição matemática de interseção para definirmos a operação de interseção entre as relações. As observações feitas para a operação UNION no que diz respeito ao domínio dos atributos e ao grau da relação, também devem ser seguidas para a operação de INTERSECTION.

Pode-se representar a operação de INTERCECAO entre as relações S e R definidas acima como sendo: R S.

EXEMPLO

Para a interseção entre os nomes de professores e alunos a álgebra relacional ficaria:

NOME_PESSOA (PROFESSOR) NOME_PESSOA (ALUNO)

Resultando:

	NOME_PESSOA
1	Cleber Dutra

Figura 1.9 – Operação INTERSECTION. Fonte: Elaborada pelo autor.

Por fim, apresentamos a operação MINUS (SUBTRAÇÃO) que representa a diferença de conjunto. O resultado dessa operação, tomando-se nossas relações S e R apresentadas anteriormente, é uma relação que inclui todas as tuplas que estão em R, mas não estão em S. Matematicamente representada por: R – S. As observações feitas para a operação UNION e INTERSECTION no que diz respeito ao domínio dos atributos e ao grau da relação, também devem ser seguidas para a operação MINUS.

A álgebra relacional ficaria:

NOME_PESSOA (PROFESSOR) – NOME_PESSOA (ALUNO)

Resultando:

	NOME_PESSOA
1	Elmasri
2	Fagundes Teles
3	Ferreira
4	Navathe

Figura 1.10 – Operação MINUS. Fonte: Elaborada pelo autor.

Observe que as operações UNION e INTERSECTION são operações comutativas, ou seja:

$$R \cup S = S \cup R \text{ e } R \cap S = S\ R.$$

Dessa forma, UNION e INTERSECTION, podem ser ditas operações n-nárias aplicáveis a qualquer número de relações, pois ambas são operações associativas, isto é:

$$R \cup (S \cup T) = (R \cup S) \cup T \text{ e } (R \cap S) = (R \cap S)\ T.$$

O mesmo não se pode dizer da operação MINUS, ou seja, ela não é comutativa, assim:

$$R - S \neq S - R.$$

1.1.3.3 Junção

A operação de JUNÇÃO representado simbolicamente na álgebra relacional pelo símbolo $\bowtie$, definindo uma junção natural ou pelo símbolo θ, definindo uma θ-junção ou equijunção. Para representar uma junção natural a relação é escrita $R \bowtie S$ onde R e S são relações. O resultado de uma junção natural é uma tabela com as combinações entre as tuplas R e S, combinadas entre si, por meio de uma coluna com os nomes em comum.

Com a θ-junção podemos combinar tuplas de duas relações R e S, por exemplo, em que a ondição de combinação entre as relações não é simplesmente uma condição de igualdade. A θ-junção pode ser descrita como: $R \underset{a\ \theta\ b}{\bowtie} S$

onde a e b são nomes de atributos e θ é uma relação binária no conjunto $\{<, \leq, =, >, \geq\}$.

Por exemplo, para as tabelas abaixo teremos o resultado de uma Junção natural $R \bowtie S$

	PROFESSOR	DISCIPLINA
1	Cleber Dutra	Matemática
2	Eduardo Tomaz	História
3	Fagundes Teles	Ciências

Figura 1.11 – Tabela PROFESSOR. Fonte: Elaborada pelo autor.

	ALUNO	DISCIPLINA
1	Cleber Dutra	Historia
2	Hernesto	Ciencias
3	Jonatan	Matematica
4	Judas	Ciencias

Figura 1.12 – Tabela ALUNO. Fonte: Elaborada pelo autor.

Como resultado de uma Junção Natural temos:

DISCIPLINA (PROFESSOR) $\bowtie$ Disciplina (ALUNO)

Resultando:

	ALUNO	DISCIPLINA	PROFESSOR
1	Cleber Dutra	Historia	Eduardo Tomaz
2	Hernesto	Ciencias	Fagundes Teles
3	Jonatan	Matematica	Cleber Dutra
4	Judas	Ciencias	Fagundes Teles

Figura 1.13 – Disciplina (PROFESSOR) $\bowtie$ Disciplina (ALUNO). Fonte: Elaborada pelo autor.

Como exemplo de uma θ-junção, consideremos as tabelas:

	PROFESSOR	IDADE_PROFESSOR
1	Cleber Dutra	20
2	Eduardo Tomaz	30
3	Fagundes Teles	40

Figura 1.14 – Tabela PROFESSOR. Fonte: Elaborada pelo autor.

	PROFESSOR	IDADE_ALUNO
1	Hernesto	16
2	Homero	35
3	Jonatan	15
4	Judas	55

Figura 1.15 – Tabela ALUNO. Fonte: Elaborada pelo autor.

Para a junção onde IDADE_ALUNO > IDADE_PROFESSOR, podemos representá-la por:
Resultando:

	ALUNO	DISCIPLINA	PROFESSOR	IDADE_PROFESSOR
1	Homero	35	Cleber Dutra	35
2	Homero	35	Eduardo Tomaz	35
3	Judas	55	Cleber Dutra	55
4	Judas	55	Eduardo Tomaz	55
5	Judas	55	Fagundes Teles	55

Figura 1.16 – θ -JUNÇÃO. Fonte: Elaborada pelo autor.

ATIVIDADES

01. Defina chave primária, chave estrangeira e qual a importância desses atributos em um modelo relacional.

02. Na tabela abaixo, qual seria uma possível chave primária? Diga os motivos que levaram a sua escolha.

NOME DA COLUNA	TIPOS DE DADOS	PERMITIR NUL...
NOME	nchar(50)	☐
SOBRENOME	nchar(50)	☐
TELEFONE	nchar(8)	☑
ENDERECO	nchar(100)	☑
CPFPAI	nchar(11)	☐
CPFMAE	nchar(11)	☐
RG	nchar(15)	☐
CPF	nchar(10)	☐

03. Defina: Tupla, Relação, Entidade, Atributo e Domínio.

04. Em uma possível importação de dados de um arquivo do Excel, o que seria necessário fazer, caso a chave primária de uma tabela fosse definida como sendo o CPF?

NOME	SOBRENOME	CPFPAI	CPFMAE	RG	CPF
ERNESTO	PAGLIA	324.522.437-98	232.234.123-09	M2.323.210	665.854.455-70
PAULA	CALABRIA	342.534.122-09	622.192.812-87		387.946.353-07
JUCA	CHAVES	341.534.122-10	123.172.129-81	8.293.232	139.128.612-00
BRAGA	ANTUNES	242.534.122-11	342.182.187-12	8.239.238	179.453.260-99
ANTUNES	BRAGA	142.534.122-12			571.972.952-62
JOAQUIM	JOSE	832.234.847-19	212.112.091-00		018.675.567-82
RUBENS	BARTOLOMEU				

05. Por que tuplas repetidas não são permitidas em uma relação?

REFERÊNCIAS BIBLIOGRÁFICAS

[1] ELMASRI, R.; NAVATHE, S., **Sistemas de Banco de Dados**. Pearson Education do Brasil, 4ª.Ed 2005.

[2] PINHEIRO, Álvaro Farias, **Fundamentos de Engenharia de Software:** Introdução a Banco de dados, Volume II, 5ª Ed 2015.

2

Linguagem SQL

2. Linguagem SQL

Com o surgimento do modelo relacional de dados, foi necessário o desenvolvimento de uma linguagem adequada para a manipulação desses dados. Na década de 70, com os estudos de Codd, um pesquisador da IBM, surgiu a linguagem SQL, sigla inglesa que quer dizer *Structured Query Language*, traduzindo livremente, Linguagem de Consulta Estruturada. O sucesso da linguagem foi tamanho que obrigou o ANSI (*American National Standarts Institute*) a padronizar a linguagem.

Em 1982, foi lançada a primeira versão oficial da SQL, outras revisões surgiram: 1987, 1992, 1999 e 2003.

É uma linguagem não procedural (não depende de *procedures*, rotinas, para ser executada) e bastante similar ao inglês, facilitando seu aprendizado.

O SQL pode ser dividido em 5 categorias:

• ***Data Definition Language*** (DDL): responsável pelos comandos de criação, alteração e manipulação das tabelas. Alguns comandos: CREATE, DROP, ALTER;

• ***Data Manipulation Language*** (DML): responsável pela criação, alteração e manutenção dos dados. Alguns comandos DML: INSERT, UPDATE, DELETE;

• ***Data Query Language*** (DQL)[1]: responsável pela pesquisa de dados. Basicamente tem-se o comando SELECT;

• ***Data Transaction Language*** (DTL): responsável pela integridade de transações dos dados. Alguns comandos: SAVEPOINT, ROLLBACK, COMMIT;

• ***Data Control Language*** (DCL): responsável pela permissão e autorização a acesso aos dados. Alguns comandos DCL: GRANT, REVOKE;

Nos parágrafos a seguir iremos estudar os principais comandos desses 5 grupos e iremos ressaltar algumas diferenças entre os principais bancos de dados existentes (Microsoft SQL Server, Oracle SQL Server, MySQL, Postgre e Firebird).

OBJETIVOS

- Conhecer a linguagem SQL;
- Aprender os comandos DDL (Linguagem de Definição de Dados);
- Aprender os comandos DML (Linguagem de Manipulação de Dados).

1 DQL tem apenas o comando SELECT, mas o SELECT também faz parte do grupo DML.

2.1 Linguagens de Definição de Dados (DDL)

Um comando DDL permite ao desenvolvedor criar e manter tabelas e outros tipos de elementos (indexes, chaves etc.). A maioria dos bancos de dados de SQL comerciais tem extensões proprietárias no DDL.

Os comandos básicos da DDL são:

CREATE: cria um objeto (uma Tabela, por exemplo) dentro da base de dados.

DROP: apaga um objeto do banco de dados.

Outros comandos DDL:

CREATE TABLE

CREATE INDEX

CREATE VIEW

ALTER TABLE

ALTER INDEX

DROP INDEX

DROP VIEW

EXEMPLO

Criação de uma tabela de DEPARTAMENTO com os campos NUMERO (inteiro) usado como chave primária (valor único que vai identificar a linha da tabela), NOME (30 caracteres) e FATOR (número com 2 casas decimais):

```
--SQLServer, Postgree, MySQL, Firebird e Oracle:
CREATE TABLE DEPARTAMENTO
(NUMERO INTEGER,
NOME VARCHAR(30) NOT NULL,
FATOR DECIMAL (3,2),
PRIMARY KEY (NUMERO));
```

2.1.1 Criação de tabela: CREATE

O comando CREATE é usado principalmente para criar tabelas. Mas também tem uma infinidade de utilidades na criação de outros objetos de um banco de dados. Pode criar, DATABASE, INDEX, USER, PROCEDURES e vários outros.

Essencialmente, para a criação de tabelas no MSSQL, sua sintaxe é:

```
CREATE TABLE
  [ database_name . [ schema_name ] . | schema_name . ] table_name
  [ AS FileTable ]
  ( { <column_definition> | <computed_column_definition>
      | <column_set_definition> | [ <table_constraint> ]
| [ <table_index> ] [ ,...n ] } )
  [ ON { partition_scheme_name ( partition_column_name ) | filegroup
    | "default" } ]
  [ { TEXTIMAGE_ON { filegroup | "default" } ]
  [ FILESTREAM_ON { partition_scheme_name | filegroup
    | "default" } ]
  [ WITH ( <table_option> [ ,...n ] ) ]
[ ; ]
```

Para criar uma tabela no ORACLE:

***create_table*::=**

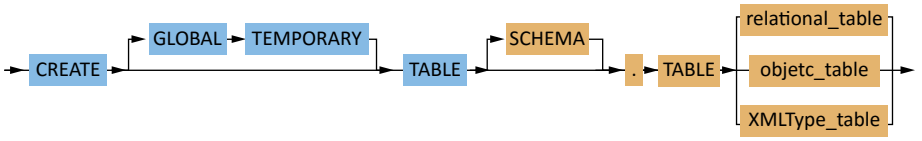


Figura 2.1 – CREATE TABLE ORACLE. Fonte: documentacao oracle, disponivel em: <http:// docs.oracle.com/cd/B19306_01/server.102/b14200/statements_7002.htm>.

Para criar uma tabela no MySQL:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
  (create_definition,...)
  [table_options]
  [partition_options]

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
  [(create_definition,...)]
  [table_options]
  [partition_options]
  select_statement
```

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] ***tbl_name***
{ LIKE ***old_tbl_name*** | (LIKE ***old_tbl_name***) }

create_definition:
col_name column_definition
| [CONSTRAINT [***symbol***]] PRIMARY KEY [***index_type***] (***index_col_name***,...)
[***index_option***] ...
| {INDEX|KEY} [***index_name***] [***index_type***] (***index_col_name***,...)
[***index_option***] ...
| [CONSTRAINT [***symbol***]] UNIQUE [INDEX|KEY]
[***index_name***] [***index_type***] (***index_col_name***,...)
[***index_option***] ...
| {FULLTEXT|SPATIAL} [INDEX|KEY] [***index_name***] (***index_col_name***,...)
[***index_option***] ...
| [CONSTRAINT [***symbol***]] FOREIGN KEY
[***index_name***] (***index_col_name***,...) ***reference_definition***
| CHECK (***expr***)

Podemos concluir, que apesar de cada gerenciador de banco de dados ter sua própria sintaxe de criação de tabelas, basicamente o comando CREATE TABLE faz:

```
CREATE TABLE table_name
(column_name1 data_type(size),
column_name2 data_type(size),
column_name3 data_type(size),
....);
```

EXEMPLO

```
--SQLServer, Postgree, MySQL, Firebird e Oracle:
CREATE TABLE DEPARTAMENTO
(NUMERO INTEGER,
NOME VARCHAR(30) NOT NULL,
FATOR DECIMAL (3,2),
PRIMARY KEY (NUMERO));
```

2.1.2 Alteração de Tabela: ALTER

Por algum motivo, após a criação de uma tabela, pode ser necessário alterá-la. Pode-se querer criar uma coluna nova, ou alterar o tamanho dos dados armazenado nessa coluna. Para isso temos o comando ALTER DATABASE.

Para adicionar uma coluna:

```
ALTER TABLE table_name
ADD column_name datatype
```

Para deletar uma coluna:

```
ALTER TABLE table_name
DROP COLUMN column_name
```

Para mudar o tipo de dados de uma coluna (no SQL Server):

```
ALTER TABLE table_name
ALTER COLUMN column_name datatype
```

Para mudar o tipo de dados de uma coluna (no Oracle versão anterior à 10G e MySQL):

```
ALTER TABLE table_name
MODIFY COLUMN column_name datatype
```

No Oracle versão superior à 10G:

```
ALTER TABLE table_name
MODIFY column_name datatype
```

2.1.3 Exclusão da Tabela: DROP

O comando DROP é usado para deletar TABELAS e também outros objetos (DATABASE, INDEX, USER, ROLE, SCHEMA, PROCEDURE etc.).

Para deletar uma tabela:

```
DROP TABLE table_name
```

2.2 Linguagem de Manipulação de Dados (DML)

Os comandos para manipulação de dados (inserir, alterar, deletar e até mesmo mostrar) são usados, como seu próprio nome diz, para manipular os dados de um banco de dados. Em desenvolvimento verão bastante o termo CRUD. Que é o acrônimo para CREATE (no sentido de criar dados), READ, UPDATE e DELETE de uma aplicação. São as operações básicas que uma aplicação na maioria das vezes deve fazer. Criar os dados, ler esses dados, atualizá-los e deletá-los, essas são as principais ações encontradas em uma aplicação. E para essas ações existem os comandos de manipulação de dados.

2.2.1 Inserção de dados: INSERT

O comando INSERT é usado para inserir ou popular uma tabela. Sua sintaxe básica é:

```
INSERT INTO table_name
VALUES (value1,value2,value3,...);
```

Ou pode-se ainda especificar quais colunas irão ser populadas:

```
INSERT INTO table_name (column1,column2,column3,...)
VALUES (value1,value2,value3,...);
```

Exemplos para o comando INSERT de uma tabela nas diversas linguagens:

– SQLServer, MySQL, Firebird

```
INSERT INTO ALUNO (NOME)
VALUES ('JOSE SILVA');
```

– Postgre

```
INSERT INTO ALUNO (MATRICULA, NOME)
VALUES (nextval('MATRICULASEQ'),'JOSE SILVA');
```

– Oracle

```
INSERT INTO ALUNO (MATRICULA, NOME)
VALUES (MATRICULASEQ.NextVal,'JOSE SILVA');
```

– Todos

```
INSERT INTO ALUNO (MATRICULA, NOME)
VALUES (10,'JOSE SILVA');
```

2.2.2 Atualização de registros: UPDATE

Usado para alteração dos dados de uma tabela ou registro.

Pode-se alterar uma linha (registro), várias linhas (registros), fazer uma alteração proveniente de uma subconsulta ou alterar registros com base em outra tabela.

```
UPDATE table_name
SET column1=value1,column2=value2,...
WHERE some_column=some_value;
```

A cláusula WHERE serve para restringir o efeito de uma coluna, se ela existir a alteração somente será válida para os registros que satisfizerem a condição. Se ela não existir todos os registros da tabela serão alterados.

Exemplos para o comando UPDATE de uma tabela nas diversas linguagens:

```
--SQLServer, Postgre, Oracle, MySQL, Firebird
UPDATE ALUNO
SET MATRICULA = 1,
NOME = 'JOSUE JOSE',
ALTURA = 1.80,
NASCIMENTO ='01/07/80'[1],
APELIDO =NULL
WHERE MATRICULA = 100;
```

2.2.3 Exclusão de Registros: DELETE

O comando DELETE é usado para excluir registros (linhas) de uma tabela.

Sua sintaxe padrão é:

```
DELETE FROM table_name
WHERE some_column=some_value;
```

EXEMPLO

```
DELETE FROM PECA
```

1 Quando se trabalha com datas, deve-se ter cuidado ao inserir uma data como texto. Deve-se verificar qual é a configuração de país do servidor. Na maioria das vezes é melhor inserir os dados no formato americano 'yyyy-mm-dd', garantindo assim a correta armazenagem do valor.

Esse comando irá apagar todos os dados da tabela.

Você pode querer apagar somente um dado específico, ou parte dos dados de uma tabela, usando o where:

```
DELETE FROM PECA where ID = 10
```

2.3 Comando SELECT

O comando SELECT é um dos, se não for o, mais usado em um sistema envolvendo banco de dados. Ele é usado para recuperar os dados armazenados. Vai ser usado bastante durante a confecção de um sistema e também nas rotinas responsáveis pela recuperação de dados desse sistema.

2.3.1 Sintaxe básica

```
SELECT      column_name [[AS] column_alias],
            column_name [[AS] column_alias], ...
            FROM table_name;
```

Você pode querer recuperar parte das colunas de uma tabela, especificando cada coluna a ser exibida ou simplesmente usar o caractere asterisco (*) recuperando assim todas as colunas da tabela.

```
SELECT * FROM table_name;
```

Você pode, ainda, para cada coluna recuperada usar um "alias" ou seja, renomear sua coluna para a consulta ser apresentada com maior clareza.

Por exemplo, na tabela abaixo de produtos (PRODUCTS[2]) você pode querer um resultado com os nomes de colunas em Português:

PRODUCTID	PRODUCTNAME
1	Chai
2	Chang
3	Aniseed Syrup
4	Chef Anton's Cajun Seasoning
5	Chef Anton's Gumbo Mix

Figura 2.2 – Tabela products. Fonte: Elaborada pelo autor.

2 A tabela *Products e Categories* fazem parte do banco de dados NORTHWIND usado pela Microsoft para exemplificar várias situações encontradas em um SGBD. (você poderá encontrá-lo em: https://www.microsoft.com/en-us/download/details.aspx?id=23654. Acessado em: 20/03/2016.

```
select ProductID as codigo,
ProductName as NomeDoProduto
from Products
```

O designador de "alias" AS, pode ser omitido (e na maior parte desse capítulo ele o será), a consulta acima ficará:

```
select ProductID codigo,
ProductName NomeDoProduto
from Products
```

Resultando:

CODIGO	NOME_DO_PRODUTO
1	Chai
2	Chang
3	Aniseed Syrup
4	Chef Anton's Cajun Seasoning
5	Chef Anton's Gumbo Mix

Figura 2.3 – Uso do alias no select. Fonte: Elaborada pelo autor.

Você pode ainda querer deixar mais visual o nome de suas colunas resultantes, deixando espaços ou até mesmo querendo usar uma palavra reservada[3]. Tente fazer um SELECT tentando dar o nome da coluna usando uma palavra reservada do SQL, por exemplo o SELECT:

```
select count(TIPO_PECA) AS SELECT
FROM PECA
WHERE TIPO_PECA = 'ELETRONICO'
```

```
Mensagem 156, Nível 15, Estado 1, Linha 1
Sintaxe incorreta próxima à palavra-chave 'SELECT'.
```

Para contornar esse "erro" basta usar [ ] ("colchetes") no MSSQL, " " ("aspas") no Oracle, MySQL e PostgreSQL, e ainda ' ' ("aspas simples") no PostgreSQL, envolvendo o nome do "alias":

3 Uma palavra reservada é uma palavra que não pode ser utilizada (na maioria das linguagens de programação) como um identificador por ser reservada para uso da gramática da linguagem.

```
select count(TIPO_PECA) AS [SELECT]
FROM PECA
WHERE TIPO_PECA = 'ELETRONICO'
```

Ou no caso de querer um ALIAS com espaços, por exemplo "QUANTIDADE DE PECAS":

```
select count(TIPO_PECA) AS [QUANTIDADE DE PECAS]
FROM PECA
WHERE TIPO_PECA = 'ELETRONICO'
```

Um "ALIAS" é muito útil em consultas onde são executadas funções que totalizam valores, tais como SUM() e COUNT(), pois quando elas aparecem em um SELECT se não for usado um "ALIAS" a coluna poderá ficar sem nome (no caso do MSSQL Server) . Por exemplo:

```
select count(TIPO_PECA)
FROM PECA
WHERE TIPO_PECA = 'ELETRONICO'
```

Resultaria:

	(Nenhum nome de coluna)
1	10

Figura 2.4 – Select sem "lias". Fonte: Elaborada pelo autor.

Com "ALIAS":

```
select count(TIPO_PECA) AS QUANTIDADE
FROM PECA
WHERE TIPO_PECA = 'ELETRONICO'
```

	QUANTIDADE
1	10

Figura 2.5 – Select com Alias. Fonte: Elaborada pelo autor.

Em alguns SGBDs (como o ORACLE e MySQL) o cabeçalho da coluna pode vir preenchido com a própria função usada. Mas sempre é mais legível dar um "nome" ao resultado que você deseja, assim sua consulta será mais fácil de ser entendida.

Exemplo de um SELECT sem ALIAS no ORACLE:

```
select customerid, COUNT(*) FROM CUSTOMERS GROUP BY customerid
```

	customerid	COUNT(*)
▪	SAVEA	31
	ERNSH	30
	QUICK	28
	HUNGO	19
	FOLKO	19
	BERGS	18
	RATTC	18

Figura 2.6 – Select count(*) sem alias no oracle. Fonte: Elaborada pelo autor.

O comando SELECT recupera os dados de uma ou várias *entidades* (tabelas), sendo um dos comandos mais simples, mas também um dos mais extensos da linguagem SQL, devido as suas funções, operandos, comandos, subcomandos e cláusulas não obrigatórias.

2.3.2 Consultas com operadores lógicos e de comparação

Para restringir ou filtrar melhor os dados a serem recuperados, podemos usar a cláusula WHERE com operadores comparativos, = (igual), > (maior), < (menor), >= (maior ou igual), <= (menor ou igual), <> (diferente), BETWEEN (entre um *range* de valores inclusivos), LIKE (semelhante ao =, mas com algumas particularidades) e ainda IN (para filtrar uma lista de valores).

Operador de negação, NOT (usado para trazer o contrário da operação lógica, por exemplo NOT =, não igual, ou o mesmo que <>, diferente.

Operadores lógicos, OR (ou), AND (e). São combinados com os operadores de comparação para ampliar as opções de busca para vários campos ou opções.

E os operadores IS NULL e IS NOT NULL, que verificam a existência de valores NULL em uma coluna.

Vejamos alguns exemplos usando a tabela CATEGORIES:

	CategoryID	CategoryName	Description
1	1	Beverages	Soft drinks, coffees, teas, beers, and ales
2	2	Condiments	Sweet and savory sauces, relishes, spreads, and...
3	3	Confections	Desserts, candies, and sweet breads
4	4	Dairy Products	Cheeses

5	5	Grains/Cereals	Breads, crackers, pasta, and cereal
6	6	Meat/Poultry	Prepared meats
7	7	Produce	Dried fruit and bean curd
8	8	Seafood	Seaweed and fish

Figura 2.7 – Tabela categories. Fonte: Elaborada pelo autor.

- =,>.>=

```
Select * from Categories where categoryid = 1
```

	CategoryID	CategoryName
1	1	Beverages

Figura 2.8 – Exemplo de select. Fonte: Elaborada pelo autor.

```
Select * from Categories where categoryid > 5
```

	CategoryID	CategoryName	Description
1	6	Meat/Poultry	Prepared meats
2	7	Produce	Dried fruit and bean curd
3	8	Seafood	Seaweed and fish

Figura 2.9 – Exemplo de Select. Fonte: Elaborada pelo autor.

```
Select * from Categories where categoryid >= 5
```

	CategoryID	CategoryName	Description
1	5	Grains/Cereals	Breads, crackers, pasta, and cereal
2	6	Meat/Poultry	Prepared meats
3	7	Produce	Dried fruit and bean curd
4	8	Seafood	Seaweed and fish

Figura 2.10 – Exemplo de select. Fonte: Elaborada pelo autor.

- **Between**

```
Select * from Categories where categoryid BETWEEN 1 AND 5
```

	CategoryID	CategoryName	Description
1	1	Beverages	Soft drinks, coffees, teas, beers, and ales
2	2	Condiments	Sweet and savory sauces, relishes, spreads, and...
3	3	Confections	Desserts, candies, and sweet breads
4	4	Dairy Products	Cheeses
5	5	Grains/Cereals	Breads, crackers, pasta, and cereal

Figura 2.11 – Exemplo de Select. Fonte: Elaborada pelo autor.

• **Like**: Como dito, o Like é se semelhante ao =, mas possui alguns caracteres "curingas", como o "%" que basicamente substitui sua posição por qualquer cadeia de caracteres.

```
Select * from Categories where CategoryName like 'Beverages'
```

Semelhante a:

```
Select * from Categories where CategoryName = 'Beverages'
```

	CategoryID	CategoryName	Description
1	1	Beverages	Soft drinks, coffees, teas, beers, and ales

Figura 2.12 – Exemplo de select. Fonte: Elaborada pelo autor.

Mas podemos usar o like com o curinga % da seguinte forma:

```
Select * from Categories where CategoryName like 'Con%'
```

O "%" diz para o SQL trazer qualquer "CategoryName" que comece com "Con" não importando qual seriam os demais caracteres.

	CategoryID	CategoryName	Description
1	2	Condiments	Sweet and savory sauces, relishes, spreads, and...
2	3	Confections	Desserts, candies, and sweet breads

Figura 2.13 – Exemplo de Select. Fonte: Elaborada pelo autor.

Outros caracteres curinga usados com o LIKE:

CARACTERE CURINGA	DESCRIÇÃO	EXEMPLO
%	Qualquer cadeia de zero ou mais caracteres.	WHERE title LIKE '%computer%' localiza todos os títulos de livro com a palavra 'computer' em qualquer lugar no título do livro.
_ (SUBLINHADO)	Qualquer caractere único.	WHERE au_fname LIKE '_ean' localiza todos os nomes de quatro letras que terminam com ean (Dean, Sean e assim por diante).
[ ]	Qualquer caractere único no intervalo ([a-f]) ou conjunto ([abcdef]) especificado.	WHERE au_lname LIKE '[C-P]arsen' localiza os sobrenomes de autores que terminem com arsen e que comecem com qualquer caractere único entre C e P, por exemplo, Carsen, Larsen, Karsen e assim por diante. Em pesquisas de intervalo, os caracteres incluídos no intervalo podem variar de acordo com as regras de classificação do agrupamento.
[^]	Qualquer caractere único que não esteja no intervalo (^[a-f]) ou conjunto ([^abcdef]) especificado.	WHERE au_lname LIKE 'de[^l]%' localiza todos os sobrenomes de autor que comecem com de e a letra seguinte não seja l.

Figura 2.14 – Caracteres Curingas. Fonte: Microsoft, diponível em: https://Msdn.microsoft.com/Pt-Br/Library/Ms179859(V=SQL.120).Aspx Acessado Em: 26/04/2016.

- **In**

```
Select * from Categories where categoryid IN (1, 2, 5, 7)
```

	CategoryID	CategoryName	Description
1	1	Beverages	Soft drinks, coffees, teas, beers, and ales
2	2	Condiments	Sweet and savory sauces, relishes, spreads, and...
3	5	Grains/Cereals	Breads, crackers, pasta, and cereal
4	7	Produce	Dried fruit and bean curd

Figura 2.15 – Exemplo de Select. Fonte: Elaborada pelo autor.

- **Not**

```
Select * from Categories where categoryid NOT IN (1, 2, 5, 7)
```

	CategoryID	CategoryName	Description
1	3	Confections	Desserts, candies, and sweet breads
2	4	Dairy Products	Cheeses
3	6	Meat/Poultry	Prepared meats
4	8	Seafood	Seaweed and fish

Figura 2.16 – Exemplo de Select. Fonte: Elaborada pelo autor.

- **Is Null E Is Not Null**

Um outro operador bastante útil é o "IS NULL". Com ele pode-se buscar qualquer valor "NULL" existente em uma coluna. O valor NULL pode ser um problema em algumas consultas do SQL, as funções de agregação, SUM(), AVG(), COUNT() etc., ignoram valores nulos, com exceção do COUNT(*). Por NULL, entende-se ausência de valor, e até por esse motivo não existe a algo como "Where campo = NULL". Para isso use o "Where campo IS NULL" ou, se o intuito é retornar apenas valores não nulos, use "WHERE campo IS NOT NULL"

Como exemplo, temos a seguinte tabela com alguns valores nulos:

	ProductID	ProductName	UnitPrice	UnitsOnOrder	UnitsInStock
1	1	Chai	18,00	NULL	NULL0
2	2	Chang	19,00	40	13
3	3	Aniseed Syrup	10,00	70	

Figura 2.17 – Tabela produtos. Fonte: Elaborada pelo autor.

```
SELECT COUNT(UnitsOnOrder) FROM PRODUTOS
```

Retorna o valor 2 (não conta o null)

```
SELECT COUNT(*) FROM PRODUTOS
```

Retorna o valor 3 (conta todos os registros)

```
SELECT SUM(UnitsInStock) FROM PRODUTOS
```

Retorna o valor 13 (soma 13 + 0)

```
SELECT AVG(UnitsInStock) FROM PRODUTOS
```

Retorna o valor 6 (média de (13 + 0) / 2). Note que se fosse levar em conta os 3 registros, a média seria 4 (média de (13+ 0) / 3)

```
SELECT * FROM PRODUTOS where UnitsInStock is null
```

Retorna o registro de ID = 1 (onde UnitsInStock é nulo)

```
SELECT * FROM PRODUTOS where UnitsInStock = 0
```

Retorna o registro de ID = 2 (onde UnitsInStock é zero)

```
SELECT * FROM PRODUTOS where UnitsOnOrder IS NOT NULL
```

Retorna os registros de ID = 2 e ID = 3 (onde UnitsOnOrder não é nulo)

2.3.3 Comandos com expressões

É possível usar expressões aritméticas junto aos comandos de SELECT do SQL.

Além das expressões básicas de adição (+), subtração (-), multiplicação (*) e divisão (/) existem várias outras:

Módulo ou resto da divisão: % no SQL SERVER e MOD no Oracle, MySQL.

Funções trigonométricas também são possíveis de serem usadas nos comandos SQL, seno, cosseno, tangente, arco seno etc.

Alguns exemplos:

Você pode usar o SQL como uma calculadora (alguns dos SGBDs permitem esse uso direto, tais como MSSQL e MySQL, já outros como o Oracle não permitem o uso direto, ele sempre espera uma cláusula FROM junto ao SELECT, para o Oracle teríamos que usar uma tabela especial chamada DUAL) para qualquer expressão matemática que queira:

- Raiz quadrada de 9:

  `select sqrt(9)` no oracle: `select sqrt(9) from DUAL`
- Três elevado ao quadrado:

  `select power(3,2)` no oracle: `select power(3,2) from DUAL`
- Qualquer conta ou expressão:

  `select ((3.0+5.0)*4.0)/3.0`

Veja as principais funções matemáticas na tabela a seguir:

FUNÇÃO	DESCRIÇÃO	EXEMPLO	RESULTADO
abs (x)	valor absoluto	abs(–17.4)	17,40
cbrt(dp)	raiz cúbica	cbrt(27.0)	3,00
ceil(dp ou numeric)	o menor inteiro não menor que o argumento	ceil(–42.8)	–42,00
ceiling(dp ou numeric)	o menor inteiro não menor que o argumento (o mesmo que ceil)	ceiling(–95.3)	–95,00
degree (dp)	radianos para graus	degrees(0.5)	286478897565412,00
exp(dp ou numeric)	exponenciação	exp(1.0)	271828182845905,00
floor(dp ou numeric)	o menor inteiro não menor que o argumento	floor(–42.8)	–43,00
ln(dp ou numeric)	logaritmo natural	ln(2.0)	0,69
log(dp ou numeric)	logaritmo na base 10	log(100.0)	2,00
log(b numeric, x numeric)	logaritmo na base b	log(2.0, 64.0)	60000000000,00
mod(y, x)	resto de y/x	mod(9,4)	1,00
pi()	constante "r"	pi ( )	314159265358979,00
power(a dp, b dp)	a elevado a b	power(9.0, 3.0)	729,00
power (a numeric, b numeric)	a elevado a b	power(9.0, 3.0)	729,00
radians (dp)	graus para radianos	radians (45.0)	0,79
random()	valor randômico entre 0.0 e 1.0	random( )	
round(dp ou numeric)	arredondar para o inteiro mais próximo	round(42.4)	42,00
round (v numeric, s integer)	arredondar para as casas decimais	round(42.4382, 2)	42,44
setseed(dp)	define a semente para as próximas chamadas a random ( )	setseed(0.54823)	1177314959,00
sign(dp ou numeric)	sinal do argumento (–1, 0, +1)	sign(–8.4)	–1,00
sqrt(dp ou numeric)	raiz quadrada	sqrt(2.0)	14142135623731,00
trunc(dp ou numeric)	trunca em direção a zero	trunc(42.8)	42,00
trunc(v numeric, s integer)	trunca com s casas decimais	trunc(42.4382, 2)	42,43
width_bucket(operando numeric, b1numeric, b2numeric, cortadorinteger	retorna a barra à qual o operando seria atribuído, em um histograma e quidepth com contador barras, um limite superior de b1, e um limite inferior de b2	width_bucket(5.35, 0.024, 10.06, 5)	3,00

Figura 2.18 – Funções matemáticas. Fonte: sourceforge.net. Disponivel em: <http://pgdocptbr.sourceforge.net/pg80/functions-math.html>.

Um comando SELECT pode agrupar dezenas de expressões de critérios e de novas tabelas. A combinação de critérios pode ser infinita e tão complexa quanto se desejar, embora a complexidade vá demandar bastante tempo de processamento do servidor. É sempre uma boa ideia mensurar essa complexidade e tentar deixar os comandos SELECTs o mais enxuto possível. A seguir a sintaxe completa do comando SELECT usada no banco de dados da Microsoft, o Microsoft SQL Server:

```
<SELECT statement> ::=
  [WITH <common_table_expression> [,...n]]
   <query_expression>
   [ ORDER BY { order_by_expression | column_position [ ASC | DESC ] }
 [ ,...n ] ]
   [ <FOR Clause>]
   [ OPTION ( <query_hint> [ ,...n ] ) ]
<query_expression> ::=
   { <query_specification> | ( <query_expression> ) }
   [ { UNION [ ALL ] | EXCEPT | INTERSECT }
     <query_specification> | ( <query_expression> ) [...n ] ]
<query_specification> ::=
SELECT [ ALL | DISTINCT ]
   [TOP ( expression ) [PERCENT] [ WITH TIES ] ]
   < select_list >
   [ INTO new_table ]
   [ FROM { <table_source> } [ ,...n ] ]
   [ WHERE <search_condition> ]
   [ <GROUP BY> ]
   [ HAVING < search_condition > ]
```

A cláusula FROM é uma das mais importantes, pois nela serão feitas as relações e expressões entre as entidades:

```
[ FROM { <table_source> } [ ,...n ] ]
<table_source> ::=
{
   table_or_view_name [ [ AS ] table_alias ] [ <tablesample_clause> ]
      [ WITH ( < table_hint > [ [ , ]...n ] ) ]
   | rowset_function [ [ AS ] table_alias ]
```

```
        [ ( bulk_column_alias [ ,...n ] ) ]
      | user_defined_function [ [ AS ] table_alias ] ]
      | OPENXML <openxml_clause>
      | derived_table [ AS ] table_alias [ ( column_alias [ ,...n ] ) ]
      | <joined_table>
      | <pivoted_table>
      | <unpivoted_table>
       | @variable [ [ AS ] table_alias ]
        | @variable.function_call ( expression [ ,...n ] ) [ [ AS ] table_alias ] [ (co-
lumn_alias [ ,...n ] ) ]
    }
    <tablesample_clause> ::=
      TABLESAMPLE [SYSTEM] ( sample_number [ PERCENT | ROWS ] )
        [ REPEATABLE ( repeat_seed ) ]

    <joined_table> ::=
    {
      <table_source> <join_type> <table_source> ON <search_condition>
      | <table_source> CROSS JOIN <table_source>
      | left_table_source { CROSS | OUTER } APPLY right_table_source
      | [ ( ] <joined_table> [ ) ]
    }
    <join_type> ::=
      [ { INNER | { { LEFT | RIGHT | FULL } [ OUTER ] } } [ <join_hint> ] ]
      JOIN

    <pivoted_table> ::=
      table_source PIVOT <pivot_clause> [ AS ] table_alias

    <pivot_clause> ::=
        ( aggregate_function ( value_column [ [ , ]...n ])
         FOR pivot_column
         IN ( <column_list> )
      )

    <unpivoted_table> ::=
      table_source UNPIVOT <unpivot_clause> [ AS ] table_alias
```

```
<unpivot_clause> ::=
    ( value_column FOR pivot_column IN ( <column_list> ) )

<column_list> ::=
     column_name [ ,...n ]
```

2.3.4 Utilização das cláusulas ORDER BY e DISTINCT

Ao retornar um conjunto de dados de um comando SELECT pode ser necessário a ordenação desses dados por qualquer uma das colunas retornadas. E também uma ordenação do menor valor, para o maior valor (ascendente) ou do maior valor para o menor valor (descendente). Para isso faz-se uso da cláusula ORDER BY ao final de um comando SELECT.

A cláusula DISTINCT garante que o retorno dos dados somente conterá valores exclusivos (distintos).

```
SELECT [ ALL | DISTINCT ]
[ TOP ( expression ) [ PERCENT ] ]
```

EXEMPLO

Na tabela PECA temos vários valores repetidos:

	TIPO_PECA	ID
1	ELETRONICO	1
2	ELETRONICO	2
3	ELETRONICO	3
4	ELETRONICO	4
5	ELETRONICO	5
6	ELETRONICO	6
7	ELETRONICO	7
8	ELETRONICO	8
9	ELETRONICO	9
10	ELETRONICO	10
11	OPTICO	11
12	OPTICO	12
13	OPTICO	13

Figura 2.19 – Tabela peça. Fonte: Elaborada pelo autor.

Podemos retornar apenas os valores “distintos” para a coluna TIPO_PECA:

```
SELECT DISTINCT TIPO_PECA FROM PECA
```

	TIPO_PECA
1	ELETRONICO
2	MECANICO
3	OPTICO

Figura 2.20 – Uso do Distinct. Fonte: Elaborada pelo autor.

A cláusula TOP retorna somente a quantidade de linhas especificadas. Muito útil quando se tem uma tabela com um grande volume de dados e você ainda não está seguro se seu SELECT está retornando os dados corretamente, pode ser usado com o PERCENT que retornará somente a porcentagem especificada de linhas com relação ao total de linhas existente na tabela:

```
SELECT top 10 TIPO_PECA FROM PECA
```

Irá retornar as 10 primeiras linhas

```
SELECT top 10 percent TIPO_PECA FROM PECA
```

Irá retornar 10 por cento de linhas de toda a tabela PECA. A Tabela PECA possui 29 linhas, 10 por cento correspondem a 3 linhas. A consulta irá retornar 3 linhas.

	TIPO_PECA
1	ELETRONICO
2	ELETRONICO
3	ELETRONICO
4	ELETRONICO
5	ELETRONICO
6	ELETRONICO
7	ELETRONICO
8	ELETRONICO
9	ELETRONICO
10	ELETRONICO

Figura 2.21 – Exemplo de select usando top 10. Fonte: Elaborada pelo autor.

Sintaxe básica da cláusula ORDE BY:

```
ORDER BY order_by_expression | column_position
  [ ASC | DESC ]
  [ ,...n ]
```

Onde:

order_by_expression

Especifica uma coluna ou expressão na qual o conjunto de resultados da consulta deve ser classificado. Uma coluna de classificação pode ser especificada como um nome ou alias de coluna, ou um inteiro não negativo que representa a posição da coluna na lista de seleção.

Várias colunas de classificação podem ser especificadas. Os nomes de coluna devem ser exclusivos. A sequência das colunas de classificação na cláusula ORDER BY define a organização do conjunto de resultados classificado. Ou seja, o conjunto de resultados é classificado pela primeira coluna e então essa lista ordenada é classificada pela segunda coluna e assim por diante.

Os nomes de coluna referenciados na cláusula ORDER BY devem corresponder a uma coluna na lista de seleção ou a uma coluna definida em uma tabela especificada na cláusula FROM sem nenhuma ambiguidade.

column_position

Você poderá designar a coluna a ser ordenada especificando a "posicao" da coluna no SELECT.

EXEMPLO

```
Select * from Categories  order by 1
```

Irá ordernar a consulta levando-se em conta a coluna de posição "1", no caso a coluna CategoryID:

	CategoryID	CategoryName	Description
1	1	Beverages	Soft drinks, coffees, teas, beers, and ales
2	2	Condiments	Sweet and savory sauces, relishes, spreads, and...
3	3	Confections	Desserts, candies, and sweet breads
4	4	Dairy Products	Cheeses
5	5	Grains/Cereals	Breads, crackers, pasta, and cereal
6	6	Meat/Poultry	Prepared meats
7	7	Produce	Dried fruit and bean curd
8	8	Seafood	Seaweed and fish

Figura 2.22 – Order by usando posicao de coluna. Fonte: Elaborada pelo autor.

A pesquisa acima tem o mesmo efeito de:

```
Select * from Categories  order by CategoryID
```

Repare que as duas pesquisas abaixo produzem ordenações diferentes:

1. `Select CategoryName, CategoryID, Description from Categories order by 2`
2. `Select * from Categories  order by 2`

ASC | DESC

Define que os valores na coluna especificada devem ser classificados em ordem crescente ou decrescente. ASC classifica do valor mais baixo para o valor mais alto. DESC classifica do valor mais alto para o valor mais baixo. ASC é a ordem de classificação padrão. Valores nulos são tratados como os menores valores possíveis.

EXEMPLO

```
Select * from Categories  order by CategoryName desc
```

	CategoryID	CategoryName	Description
1	8	Seafood	Seaweed and fish
2	7	Produce	Dried fruit and bean curd
3	6	Meat/Poultry	Prepared meats
4	5	Grains/Cereals	Breads, crackers, pasta, and cereal
5	4	Dairy Products	Cheeses
6	3	Confections	Desserts, candies, and sweet breads
7	2	Condiments	Sweet and savory sauces, relishes, spreads, and...
8	1	Beverages	Soft drinks, coffees, teas, beers, and ales

Figura 2.23 – Uso do order by. Fonte: Elaborada pelo autor.

2.3.5 Funções de Grupo, cláusulas GROUP BY e HAVING

Funções de agregação ou funções de grupo, são funções responsáveis por totalizar ou sumarizar valores de uma tabela. Executam um cálculo em um conjunto de valores e retornam um único valor. As principais são:

SUM – retorna a soma de uma coluna;
AVG - retorna a média aritmética;
MIN - retorna o valor mínimo de uma coluna;
MAX - retorna o valor máximo de uma coluna;
COUNT - retorna a quantidade de colunas;
Por exemplo, para a tabela Categories vista anteriormente, teremos:

```
Select count(*) from Categories  where categoryid
```

Irá retornar o valor 8

```
Select min(CategoryID) from Categories
```

Irá retornar o valor 1

```
Select sum(CategoryID) from Categories
```

Irá retornar o valor 36

As funções MIN, SUM, MAX não fazem muito sentidos ao serem aplicadas a uma coluna de chave primária. Fazem mais sentido quando são aplicadas em colunas onde podemos totalizar valores mais representativos, tais como 'preço', 'data de entrega' etc.

Por exemplo, no banco de dados NORTHIWIND temos uma tabela de "detalhe de pedidos" e nela se encontram os "preços" de todos os pedidos e a "quantidade" de produtos comprados, podemos querer saber qual é a média de valores dos pedidos, a média de itens pedidos, a quantidade total de itens pedidos e a média de itens pedidos:

	OrdertID	ProductID	UnitPrice	Quantity	Dicount
1	10248	11	10,00	12	0
2	10240	42	0,80	10	0
3	10248	72	34,80	5	0
4	10249	14	18,60	9	0
5	10249	51	42,40	40	0
6	10250	41	7,70	10	0
7	10250	51	42,40	35	0,15
8	10250	65	16,80	15	0,15
9	10251	22	16,80	6	0,05
10	10251	57	15,60	15	0,05
11	10251	65	16,80	20	0
12	10252	20	64,80	40	0,05
13	10252	33	2,00	25	0,05
14	10252	60	27,20	40	0
15	10253	31	10,00	20	0

Figura 2.24 – Tabela de pedidos. Fonte: Elaborada pelo autor.

```
select avg(UnitPrice) GastoMedioPorPedido, avg(quantity)
QuantidadeItensPorPedido, sum(unitprice) GastoTotal, sum(quantity)
QuantidadeTotal from [Order Details]
```

	GastoMedioPorPedido	QuantidadeItensPorPedido	GastoTotal	QuantidadeTotal
1	26,2185	23	56500,91	51317

Figura 2.25 – Exemplo de funções de agregação. Fonte: Elaborada pelo autor;

Outra cláusula bastante usada é a GROUP BY, responsável por agrupar dados de uma pesquisa, seu uso é intenso quando se quer somar ou contar membros de um determinado "grupo". Por exemplo, se quisermos saber quantos (contar) produtos são do tipo "eletrônico" em uma tabela de componentes para montagem de peças podemos usar:

```
select count(TIPO_PECA) QUANT, TIPO_PECA
FROM PECA
WHERE TIPO_PECA = 'ELETRONICO'
GROUP BY TIPO_PECA
```

Repare que a pesquisa acima pode ser escrita também como:

```
select count(TIPO_PECA) QUANT, 'ELETRONICO' TIPO_PECA
FROM PECA
WHERE TIPO_PECA = 'ELETRONICO'
```

Que não faz muito sentido, já que para cada tipo diferente, teríamos que conhecer exatamente seu nome para poder criar o resultado desejado.

Mas, e se quisermos trazer a contagem de todos os tipos existentes (eletrônicos, mecânicos, opticos etc.)?

Agora sim faz mais sentido usar a expressão GROUP BY, pois, ela irá retornar para cada linha de tipos de matéria a sua quantidade, sem precisar especificar na sua cláusula where:

```
select count(TIPO_PECA) QUANT, TIPO_PECA
FROM PECA
GROUP BY TIPO_PECA
```

Que irá retornar:

	QUANT	TIPO_PECA
1	10	ELETRONICO
2	15	MECANICO
3	4	OPTICO

Figura 2.26 – Exemplo de Group By. Fonte: Elaborada pelo autor.

A cláusula HAVING dá um poder a mais para o GROUP BY. Com ela podemos filtrar melhor os resultados de um GROUP BY ela sempre virá após um GROUP BY.

Se quisermos descobrir quais os tipos de peças que estão com uma contagem baixa de estoque (com a contagem em estoque menor do que dez unidades)?

Resposta:

```
select count(TIPO_PECA) QUANT, TIPO_PECA
FROM PECA
GROUP BY TIPO_PECA
HAVING count(TIPO_PECA) < 10
```

QUANT	TIPO_PECA
4	OPTICO

Figura 2.27 – Exemplo da cláusula having. Fonte: Elaborada pelo autor.

2.3.6 Comandos de Junção

A cláusula de junção (JOIN) é usada basicamente para retornar os dados de duas ou mais tabelas relacionadas.

A sintaxe padrão para a criação de uma consulta JOIN usando o SQL Server da Microsoft está definida abaixo e será comentada (para maiores detalhes verifique a documentação Microsoft SQL Server disponível em: <https://msdn.microsoft.com/pt-br/library/ms177634(v=SQL.120).aspx>. Acessado em: 22/03/2016).

```
[ FROM { <table_source> } [ ,...n ] ]
<table_source> ::=
{
  table_or_view_name [ [ AS ] table_alias ] [ <tablesample_clause> ]
    [ WITH ( < table_hint > [ [ , ]...n ] ) ]
  | rowset_function [ [ AS ] table_alias ]
    [ ( bulk_column_alias [ ,...n ] ) ]
  | user_defined_function [ [ AS ] table_alias ] ]
  | OPENXML <openxml_clause>
  | derived_table [ AS ] table_alias [ ( column_alias [ ,...n ] ) ]
  | <joined_table>
```

```
    | <pivoted_table>
    | <unpivoted_table>
     | @variable [ [ AS ] table_alias ]
      | @variable.function_call ( expression [ ,...n ] ) [ [ AS ] table_alias ] [ (co-
lumn_alias [ ,...n ] ) ]
  }
  <tablesample_clause> ::=
    TABLESAMPLE [SYSTEM] ( sample_number [ PERCENT | ROWS ] )
      [ REPEATABLE ( repeat_seed ) ]

  <joined_table> ::=
  {
    <table_source> <join_type> <table_source> ON <search_condition>
    | <table_source> CROSS JOIN <table_source>
    | left_table_source { CROSS | OUTER } APPLY right_table_source
    | [ ( ] <joined_table> [ ) ]
  }
  <join_type> ::=
    [ { INNER | { { LEFT | RIGHT | FULL } [ OUTER ] } } [ <join_hint> ] ]
    JOIN
```

Para melhor exemplificar vamos tomar duas tabelas[4] e analisar as diversas possibilidades para as junções:

ProductID	ProductName	CategoryFK
71	Flotemysost	4
72	Mozzarella di Giovanni	4
73	Röd Kaviar	8
74	Longlife Tofu	7
75	Rhönbräu Klosterbier	1
76	Lakkalikööri	1
77	Original Frankfurter grüne Sobe	2
78	Product Test 1	100
79	Product Test 2	101

Figura 2.28 – Tabela produto. Fonte: Elaborada pelo autor.

4 As tabelas CATEGORIA e PRODUTO foram criadas a partir dos dados das tabelas CATEGORIES e PRODUCTS do banco de dados exemplo da Microsoft chamado NORTHWIND (você poderá encontrá-lo em: https://www.microsoft.com/en-us/download/details.aspx?id=23654. Acessado em: 20/03/2016.

CategoryID	CategoryName
1	Beverages
9	Category 1
10	Category 2
2	Condiments
3	Confections
4	Dairy Products
5	Grains/Cereals
6	Meat/Poultry
7	Produce
8	Seafood

Figura 2.29 – Tabela categoria. Fonte: Elaborada pelo autor.

<joined_table>

São as tabelas a serem combinadas onde serão extraídos os dados. Podem ser usadas várias tabelas e Junções aninhados. Use parênteses para determinar a ordem de execução das Junções.

JOIN

Indica que a operação de junção especificada serão executadas entre as tabelas ou visões especificadas.

ON <search_condition>

Especifica o critério de combinação ou condição de busca a ser usado entre as tabelas, geralmente são os critérios de relacionamentos definidos entre as tabelas. Para nossas tabelas mencionadas acima, temos:

```
SELECT *
FROM
    CATEGORIA JOIN PRODUTO
        ON
            CategoryID = CategoryFK
```

Quando as condições de busca especificarem colunas entre as tabelas, essas colunas não necessariamente serão de mesmos nomes ou mesmos tipos de dados. No entanto se os tipos de dados não forem compatíveis uma conversão entre os tipos será necessária. Para alguns tipos de dados o SQL faz uma conversão "implicita" entre esses tipos de dados diferentes, ou seja, o próprio gerenciador irá "tentar" combinar os dados convertendo para um tipo de dados onde será possível executar a condição de busca. Para outros tipos, ou dependendo da necessidade do autor da busca, a conversão entre os tipos deverá ser

explícita e deverá ser usada alguma função de conversão de conversão de tipos (CAST ou CONVERT para o SQL, ou CAST para outras linguagens).

<join_type>

É o tipo de Junção buscada no resultado. A saber:

INNER

O parâmetro INNER especifica o retorno de apenas os dados correspondentes a ambas as tabelas. Esse é o parâmetro padrão e se nada for especificado será esse o padrão adotado pelo interpretador do SQL.

Tomando-se como exemplo nossas tabelas PRODUTO e CATEGORIA temos:

```
select * from CATEGORIA INNER JOIN PRODUTO ON CategoryID = CategoryFK
```

ou simplesmente:

```
select * from CATEGORIA JOIN PRODUTO ON CategoryID = CategoryFK
```

Resultando na seguinte junção:

		ProductID	ProductName	CategoryFK
1	Beverages	75	Rhönbräu Klosterbier	1
1	Beverages	76	Lakkalikööri	1
2	Condiments	77	Original Frankfurter grüne Sobe	2
4	Dairy Products	71	Flotemysost	4
4	Dairy Products	72	Mozzarella di Giovanni	4
7	Seafood	74	Longlife Tofu	7
8	Produce	73	Röd Kaviar	8

Figura 2.30 – Exemplo de junção com inner. Fonte: Elaborada pelo autor.

FULL [OUTER]

O parâmetro FULL especifica o retorno de todos os dados de ambas as tabelas independente da correspondência entre elas, as colunas que são correspondentes são preenchidas, as que não são correspondentes virão preenchidas com NULL. Para maior clareza pode vir seguido da palavra OUTER.

EXEMPLO

```
select * from CATEGORIA FULL OUTER JOIN PRODUTO ON CategoryID = CategoryFK
```

ou:

```
select * from CATEGORIA FULL JOIN PRODUTO ON CategoryID = CategoryFK
```

Resultando:

CategoryID	CategoryName	ProductID	ProductName	CategoryFK
1	Beverages	75	Rhönbräu Klosterbier	1
1	Beverages	76	Lakkalikööri	1
9	Category 1	NULL	NULL	NULL
10	Category 2	NULL	NULL	NULL
2	Condiments	77	Original Frankfurter grüne Sobe	2
3	Confections	NULL		
4	Dairy Products	71	Flotemysost	4
4	Dairy Products	72	Mozzarella di Giovanni	4
5	Grains/Cereals	NULL	NULL	NULL
6	Meat/Poultry	NULL	NULL	NULL
7	Produce	74	Longlife Tofu	7
8	Seafood	73	Röd Kaviar	8
NULL	NULL	78	Product Test 1	100
NULL	NULL	79	Product Test 2	101

Figura 2.31 – Full outer join. Fonte: Elaborada pelo autor.

Repare que no comando anterior, para as categorias: Category 1, Category 2, Grains/Cereals e Meat/Poultry são inseridos valores Nulos para as colunas correspondentes da tabela PRODUTO. E para os produtos: Produtct Test1 e Product Test2 são inseridos valores Nulos para as colunas correspondentes da tabela CATEGORIA.

LEFT [OUTER]

O parâmetro LEFT especifica o retorno de todos os dados da tabela à ESQUERDA da relação além dos dados correspondentes a ambas as tabelas, as colunas que são correspondentes são preenchidas, as que não são correspondentes (à tabela da esquerda) virão preenchidas com NULL. Para maior clareza pode vir seguido da palavra OUTER.

EXEMPLO

```
select * from CATEGORIA LEFT OUTER JOIN PRODUTO ON CategoryID = CategoryFK
```

ou:

```
select * from CATEGORIA LEFT JOIN PRODUTO ON CategoryID = CategoryFK
```

Resultando:

CategoryID	CategoryName	ProductID	ProductName	CategoryFK
1	Beverages	75	Rhönbräu Klosterbier	1
1	Beverages	76	Lakkalikööri	1
9	Category 1	NULL	NULL	NULL
10	Category 2	NULL	NULL	NULL
2	Condiments	77	Original Frankfurter grüne Sobe	2
3	Confections	NULL		
4	Dairy Products	71	Flotemysost	4
4	Dairy Products	72	Mozzarella di Giovanni	4
5	Grains/Cereals	NULL	NULL	NULL
6	Meat/Poultry	NULL	NULL	NULL
7	Produce	74	Longlife Tofu	7
8	Seafood	73	Röd Kaviar	8

Figura 2.32 – Left outer join. Fonte: Elaborada pelo autor.

RIGHT [OUTER]

É praticamente o contrário de LEFT. O parâmetro RIGHT especifica o retorno de todos os dados da tabela à DIREITA da relação além dos dados correspondentes a ambas as tabelas, as colunas que são correspondentes são preenchidas, as que não são correspondentes (à tabela da diretia) virão preenchidas com NULL. Para maior clareza pode vir seguido da palavra OUTER.

EXEMPLO

```
select * from CATEGORIA RIGHT OUTER JOIN PRODUTO ON CategoryID = CategoryFK
```

ou

```
select * from CATEGORIA RIGHT JOIN PRODUTO ON CategoryID = CategoryFK
```

Resultando:

CategoryID	CategoryName	ProductID	ProductName	CategoryFK
4	Dairy Products	71	Flotemysost	4
4	Dairy Products	72	Mozzarella di Giovanni	4
8	Seafood	73	Röd Kaviar	8
7	Produce	74	Longlife Tofu	7
1	Beverages	75	Rhönbräu Klosterbier	1
1	Beverages	76	Lakkalikööri	1
2	Condiments	77	Original Frankfurter grüne Sobe	2
NULL	NULL	78	Product Test 1	100
NULL	NULL	79	Product Test 2	101

Figura 2.33 – Right outer join. Fonte: Elaborada pelo autor.

CROSS[5] JOIN

Especifica o produto carteziano entre duas tabelas. Para cada linha da tabela a esquerda é retornado todas as linhas da tabela a direita. Não terá a cláusula ON para definir critérios de busca. Se a cláusula ON for especificada, um erro de sintaxe será retornado.

EXEMPLO

```
select * from CATEGORIA CROSS JOIN PRODUTO
```

Resultando 90 linhas (9 linhas da tabela CATEGORIA X 10 linhas da tabela PRODUTO). Abaixo apenas as primeiras linhas da operação CROSS JOIN:

CategoryID	CategoryName	ProductID	ProductName	CategoryFK
1	Beverages	71	Flotemysost	4
1	Beverages	72	Mozzarella di Giovanni	4
1	Beverages	73	Röd Kaviar	8
1	Beverages	74	Longlife Tofu	7
1	Beverages	75	Rhönbräu Klosterbier	1
1	Beverages	76	Lakkalikööri	1
1	Beverages	77	Original Frankfurter grüne Sobe	2
1	Beverages	78	Product Test 1	100
1	Beverages	79	Product Test 2	101
2	Condiments	71	Flotemysost	4
2	Condiments	72	Mozzarella di Giovanni	4
2	Condiments	73	Röd Kaviar	8
2	Condiments	74	Longlife Tofu	7
2	Condiments	75	Rhönbräu Klosterbier	1
2	Condiments	76	Lakkalikööri	1
2	Condiments	77	Original Frankfurter grüne Sobe	2
2	Condiments	78	Product Test 1	100
2	Condiments	79	Product Test 2	101

Figura 2.34 – Cross join. Fonte: Elaborada pelo autor.

Aqui, cabe ainda um parênteses sobre a cláusula JOIN. A cláusula JOIN foi introduzido na versão SQL/92 no padrão ANSI. O ORACLE até as versões anteriores a 9i usava o padrão de linguagem ANSI versão SLQ/86 que não existia o JOIN definido. Mas existe uma forma de se obter o relacionamento entre a tabelas, que inclusive ainda é muito usado, sem o auxílio do comando JOIN. Basta especificar a lista de tabelas na cláusula FROM e usar as "conexões" entre as tabelas (suas chaves primárias e estrangeiras) na cláusula WHERE fazendo uma igualdade entre as colunas chaves. Exemplificando:

5 A cláusula CROSS parece não fazer muito sentido, já que o objetivo maior em um banco de dados e retornar dados relacionados. Mas pode ser útil na obtenção de uma "massa de dados" para ser usado durante fases de teste.

Para o JOIN:

```
select * from CATEGORIA JOIN PRODUTO ON CategoryID = CategoryFK
```

Podemos reescrever:

```
select * from CATEGORIA, PRODUTO WHERE CategoryID = CategoryFK
```

O resultado será o mesmo.

2.3.7 SubConsultas Aninhadas e Correlatas.

Podemos usar uma consulta dentro de uma outra consulta. Nesse caso a consulta interna é dita "instrução-contêiner". Por exemplo, se tivermos uma tabela com a codificação de todos os tipos de peças, podemos retornar esses tipos por meio de um subselect:

```
select TIPO_PECA TIPO, (SELECT COD_PECA FROM NATUREZA_PECA
WHERE COD_PECA = ID_PECA) CODIGO
FROM PECA
WHERE TIPO_PECA = 'ELETRONICO'
```

Essa subconsulta é dita correlata, pois ela faz uma "referência" à consulta mais externa.

Lembrando que ID_PECA está na tabela PECA e COD_PECA na tabela NATUREZA_PECA). Para um subselect funcionar ele não pode retornar mais de um resultado. Senão um erro será apresentado:

A subconsulta retornou mais de 1 valor. Isso não é permitido quando a subconsulta segue um =, !=, <, <= , >, >= ou quando ela é usada como uma expressão.

E caso não haja um valor correspondente entre COD_PECA e ID_PECA, o resultado correspondente para a coluna CODIGO virá preenchida com o valor NULL.

Note ainda, que a relação entre ID_PECA e COD_PECA pode ser retornada por meio de uma cláusula JOIN:

```
select TIPO_PECA, COD_PECA, ID_PECA
FROM PECA JOIN NATUREZA_PECA ON ID_PECA = COD_PECA
WHERE TIPO_PECA = 'ELETRONICO'
```

Nesse caso, pode haver mais de uma relação entre ID_PECA e COD_PECA que os dados serão retornados.

Note também, que a consulta acima pode ser escrita também da seguinte forma:

```
SELECT TIPO_PECA, COD_PECA, ID_PECA
FROM PECA, NATUREZA_PECA
WHERE ID_PECA = COD_PECA
AND TIPO_PECA = 'ELETRONICO'
```

A visualização da pesquisa acima é bastante fácil e intuitiva. Então qual a vantagem da cláusula JOIN?

A resposta está em seus argumentos LEFT, RIGHT e CROSS.

Uma subconsulta não correlata quer dizer que ela não faz referência a sua consulta mais interna. É muito usada na cláusula WHERE como sendo um grupo de dados para restrição da pesquisa. Exemplo:

```
SELECT TIPO_PECA
FROM PECA
WHERE ID_PECA = (SELECT MAX(COD_PECA) FROM NATUREZA_PECA)
```

Note que a consulta interna não faz referência a nenhuma coluna da tabela principal. Nesse tipo de pesquisa, apenas uma coluna poderá ser especificada e ainda assim poderá trazer apenas uma linha de dados.

A pesquisa abaixo não faz sentido e retornará o erro:

```
SELECT TIPO_PECA
FROM PECA
WHERE ID_PECA = (SELECT COD_PECA, VALOR_NATUREZA
FROM NATUREZA_PECA)
```

Somente uma expressão pode ser especificada na lista de seleção quando a subconsulta não é introduzida com EXISTS.

E ao tentar corrigir, deixando apenas um argumento ela ainda poderá retornar um erro:

```
SELECT TIPO_PECA
FROM PECA
WHERE ID_PECA = (SELECT COD_PECA FROM NATUREZA_PECA)
```

A subconsulta retornou mais de 1 valor. Isso não é permitido quando a subconsulta segue um =, !=, <, <= , >, >= ou quando ela é usada como uma expressão.

Isso acontece por que o operador igual (=) está esperando apenas um valor para comparação. Você pode corrigir a consulta acima (mas apesar de não retornar um erro isso não quer dizer que ela está "logicamente" correta) usando o operador IN:

```
SELECT TIPO_PECA
FROM PECA
WHERE ID_PECA IN (SELECT COD_PECA FROM NATUREZA_PECA)
```

2.3.8 Operadores de Conjunto

Vimos no capítulo 1 algumas operações de álgebra relacional, apresentando as operações entre conjuntos possíveis, incluindo UNION (UNIÃO), INTERSECTION (INTERSEÇÃO), SET DIFFERENCE (DIFERENÇA ENTRE CONJUNTOS) e CROSS PRODUCT (PRODUTO CARTESIANO).

Essas operações permitem descobrir os dados correlatos segundo as instruções de conjuntos conhecidas, interseção, união, diferença e produto cartesiano.

```
SELECT NOME_PROFESSOR NOME_PESSOA FROM PROFESSOR
UNION
SELECT NOME_ALUNO FROM DISCIPLINA_NOTA

SELECT NOME_PROFESSOR NOME_PESSOA FROM PROFESSOR
INTERSECT
SELECT NOME_ALUNO FROM DISCIPLINA_NOTA

SELECT NOME_PROFESSOR NOME_PESSOA FROM PROFESSOR
EXCEPT
SELECT NOME_ALUNO FROM DISCIPLINA_NOTA
```

2.4 Criando Outros Objetos de Banco de Dados

Vários outros objetos são possíveis em um banco de dados. Pode-se criar suas próprias funções (por exemplo, você pode criar uma função de validação de CPF):

```
CREATE FUNCTION CPF_VALIDO(@CPF VARCHAR(11))
RETURNS CHAR(1)
AS
BEGIN
 DECLARE @IND INT,
      @SOMA INT,
      @DIG1 INT,
      @DIG2 INT,
      @CPF_TEMP VARCHAR(11),
      @DIGIT_IGUAIS CHAR(1),
      @RESULT CHAR(1)

 SET @RESULT = 'N'
 SET @CPF_TEMP = SUBSTRING(@CPF,1,1)
 SET @IND = 1
 SET @DIGIT_IGUAIS = 'S'

 WHILE (@IND <= 11)
 BEGIN
  IF SUBSTRING(@CPF,@IND,1) <> @CPF_TEMP
   SET @DIGIT_IGUAIS = 'N'
  SET @IND = @IND + 1
 END;

 IF @DIGIT_IGUAIS = 'N'
 BEGIN
  SET @SOMA = 0
  SET @IND = 1
  WHILE (@IND <= 9)
  BEGIN
   SET @SOMA = @SOMA + CONVERT(INT,SUBSTRING(@CPF,@IND,1)) *
(11 - @IND);
   SET @IND = @IND + 1
  END

  SET @DIG1 = 11 - (@SOMA % 11)
```

```
        IF @DIG1 > 9
         SET @DIG1 = 0;
        SET @SOMA = 0
        SET @IND = 1
        WHILE (@IND <= 10)
        BEGIN
          SET @Soma = @Soma + CONVERT(INT,SUBSTRING(@CPF,@IND,1)) *
(12 - @IND);
         SET @IND = @IND + 1
        END
        SET @DIG2 = 11 - (@SOMA % 11)
        IF @DIG2 > 9
         SET @DIG2 = 0;
           IF (@DIG1 = SUBSTRING(@CPF,LEN(@CPF)-1,1)) AND (@DIG2 =
SUBSTRING(@CPF,LEN(@CPF),1))
         SET @RESULT = 'S'
        ELSE
         SET @RESULT = 'N'
       END
       RETURN @RESULT
      END
```

Se você tem uma consulta que é feita várias vezes ou a usa em diversos locais de seu sistema, você pode querer criar uma "stored procedure" com essa consulta armazenada. Assim, se precisar alterar a pesquisa, poderá fazer em apenas um local, não sendo necessário procurar em todo o seu sistema e ainda correr o risco de "esquecer" algum lugar que vai chamar a consulta da maneira antiga e retornar um erro para o usuário do sistema.

Um outro uso bastante difundido de "stored procedures" é fazer toda a lógica de inserir, alterar, deletar os dados criando essas regras no próprio banco de dados. Assim você tem um controle maior de seus dados e poderá fazer a manutenção dessas operações em apenas um lugar.

Uma outra forma de consulta armazenada bastante usada, são as "VIEWS" que nada mais são, do que consultas prontas para uso, e ainda podem ser usadas em outras consultas de maneira transparente, como sendo uma tabela (ou entidade).

Vários outros objetos são possíveis e ajudam no uso de um banco de dados. Algum deles:

Schemas (ajudam a organizar um banco de dados), índices (melhoram as pesquisas retornando os dados mais rapidamente), *data types* (tipo de dados definidos pelo usuário), *rules* (regras para uso de tabelas e banco de dados), *trigger*(é como uma procedure, mas são executadas logo após um *update*, *insert* ou *delete*, dependendo de como são programadas). E muitos outros objetos.

2.4.1 Criando Visões

Mencionamos no item anterior o objeto VIEW que auxilia o sistema de banco de dados. Com ele, pode-se criar uma operação SELECT e armazená-la para posterior uso até mesmo com outras pesquisas. Ela se comporta como se fosse uma tabela de nosso Banco de Dados.

As vantagens de se utilizar VIEWS são inúmeras:

• Economizar tempo com retrabalho: Uma vez construída uma *view*, você pode utilizá-la em qualquer parte de seu sistema, exatamente como se ela fosse uma tabela;

• Velocidade de acesso às informações: Uma VIEW é guardada em um sistema de banco de dados na forma "compilada", ou seja, o seu Sistema de Banco de Dados já sabe que você vai querer usar aquele SELECT novamente e o guarda no sistema de maneira otimizada.

• Mascarar complexidade do banco de dados: Você pode criar uma VIEW apenas com as tabelas que são essenciais ao usuário, escondendo outras que podem não ser entendidas para o usuário. E até mesmo esconder colunas que podem ter valores restritos a todos os usuários do banco de dados.

Sintaxe básica para a criação de uma View:

```
CREATE VIEW [ schema_name . ] view_name
AS select_statement
[;]
```

Para exemplificar iremos usar as tabelas mencionadas anteriormente, PRODUTO e CATEGORIA. Podemos já deixar uma consulta "gravada" em um formato de VIEW caso essa relação entre PRODUTO e CATEGORIA seja bastante usada no Banco de Dados em questão.

Por exemplo:

```
CREATE VIEW CATEGORIA_PRODUTO
AS
SELECT CategoryID, ProductID, CategoryName, ProductName
    from  CATEGORIA JOIN PRODUTO
            ON CategoryID = CategoryFK;
```

Após a criação de uma VIEW seu uso é feito como se fosse uma Tabela já existente no Banco de dados:

```
SELECT * FROM CATEGORIA_PRODUTO      ORDER BY CategoryName
```

Olhando apenas o resultando fica impossível distinguir se é uma Tabela ou uma View:

CategoryID	ProductID	CategoryName	ProductName
1	75	Beverages	Rhönbräu Klosterbier
1	76	Beverages	Lakkalikööri
2	77	Confections	Original Frankfurter grüne Sobe
4	71	Dairy Products	Flotemysost
4	72	Dairy Products	Mozzarella di Giovanni
7	74	Produce	Longlife Tofu
8	73	Seafood	Röd Kaviar

Figura 2.35 – Resultado de um select a partir de uma view. Fonte: Elaborada pelo autor.

2.4.2 Criando uma sequência

Uma sequência é uma maneira muito útil de definir um objeto que irá retornar sempre um novo número que irá respeitar uma sequência pré-definida. Em alguns sistemas é usado para definir uma chave única, um índice de uma tabela.

Sintaxe:

```
CREATE SEQUENCE [schema_name . ] sequence_name
  [ AS [ built_in_integer_type | user-defined_integer_type ] ]
  [ START WITH <constant> ]
  [ INCREMENT BY <constant> ]
  [ { MINVALUE [ <constant> ] } | { NOMINVALUE } ]
  [ { MAXVALUE [ <constant> ] } | { NOMAXVALUE } ]
  [ CYCLE | { NOCYCLE } ]
  [ { CACHE [ <constant> ] } | { NO CACHE } ]
  [ ; ]
```

No MS SQL Server, para se definir uma coluna chave, é bastante usado o tipo "IDENTITY" que corresponde a uma sequência de números criadas e inseridas automaticamente em uma tabela de banco de dados.

Por exemplo, para criar uma tabela no SQL e usar um campo de auto numeração para usar como chave primária usamos:

```
--SQLServer
CREATE TABLE CARGO(
NUMERO INTEGER IDENTITY (1,1),
NOME VARCHAR(30) NOT NULL,
PRIMARY KEY (NUMERO));
```

Exemplo de uso:

```
--SQLServer / MySQL
INSERT INTO CARGO (NOME) VALUES ('ENGENHEIRO')
```

Para o Postgree ou Oracle:

```
--Postgree / Oracle
CREATE TABLE CARGO(
NUMERO INTEGER,
NOME VARCHAR(30) NOT NULL,
PRIMARY KEY (NUMERO));
CREATE SEQUENCE CARGOSEQ, INCREMENT
BY 1 START WITH 1;
```

Exemplo de uso:

```
--Postgree
INSERT INTO CARGO (NUMERO, NOME) VALUES (nextval('CARGOSEQ'),'ENGENHEIRO');
--Oracle
INSERT INTO CARGO (NUMERO, NOME) VALUES (CARGOSEQ.NextVal,'ENGENHEIRO');
```

Para o MySQL:

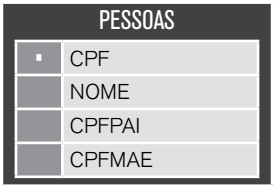
```
--MySQL
CREATE TABLE CARGO(
NUMERO INTEGER AUTO_INCREMENT,
NOME VARCHAR(30) NOT NULL,
PRIMARY KEY (NUMERO));
```

Exemplo de uso:

```
--SQLServer / MySQL
INSERT INTO CARGO (NOME) VALUES ('ENGENHEIRO')
```

ATIVIDADES

01. A relação abaixo é possível? Nela o CPF da Tabela PESSOAS está relaciona com o CPFPAI da mesma tabela PESSOAS e o CPFMAE também está relacionado com o CPF da mesma tabela.



02. Escreva uma consulta (SELECT) para retornar o Nome de uma pessoa e de seu Pai.

03. Escreva uma consulta (SELECT) para retornar o Nome de uma pessoa, seu Pai e sua Mãe.

04. Em uma instrução SELECT, retorne os dados da tabela do exercício 1 do capítulo 1 em duas colunas e no seguinte formato:

Sobrenome em maiúsculo, (com a vírgula) Nome com o primeiro caractere em Maiúsculo e, CPF (sem pontos e traços), exemplo:

PAGLIA, Ernesto 32452243798

05. Escreva a instrução INSERT para a inserção dos seguintes dados na tabela PESSOAS:

CPF: 038.053.186-00
NOME: João
SOBRENOME: Silva
CPFPAI: NULL
CPFMAE: NULL

06. Qual a diferença entre os SELECTS:

```
select ((3.0+5.0)*4.0)/3.0 e select ((3+5)*4)/3
```

07. Escreva a consulta abaixo sem usar o JOIN (use o WHERE). Qual você achou mais fácil de construir? E de entender? Justifique.

```
SELECT * FROM Orders JOIN [Order Details] ON [Order Details].OrderID = Orders.OrderID JOIN Products ON Products.ProductID = [Order Details].ProductID
```

08. Na consulta abaixo, qual é o erro? Por que? Como você reescreveria?

```
SELECT sum(quantity), ProductName FROM [Order Details], Products where [Order Details].ProductID = Products.ProductID and sum(quantity) > 10
```

09. Na sua opinião, qual a maior vantagem de se usar uma VIEW?

10. A consulta a seguir está correta? O que ela faz? Como ela poderia ser reescrita?

```
select OrderID, (select CustomerID from Customers where customerId = orders.customerId) customerId, (select [CompanyName] from Customers where customerId = orders.customerId) [CompanyName] from orders
```

REFERÊNCIAS BIBLIOGRÁFICAS

[1] ELMASRI, R.; NAVATHE, S., **Sistemas de Banco de Dados.** Pearson Education do Brasil, 4ª.Ed 2005.

[2] PINHEIRO, Álvaro Farias, **Fundamentos de Engenharia de Software**: Introdução a Banco de dados, Volume II, 5ª Ed 2015.

[3] Microsoft SQL SERVER. **Documentação Microsoft SQL Server 2014**, [Online]. Available: https://msdn.microsoft.com/library/mt590198.aspx. [Acesso em 20 03 2016].

[3] ORACLE SERVER, **Documentação ORACLE**, [Online]. Available: http://www.oracle.com/technetwork/pt/indexes/documentation/index.html. [Acesso em 20 03 2016].

[4] MySQL, **Documentação MySQL**, [Online]. Available: https://dev.MySQL.com/doc/refman/5.7/en/. [Acesso em 20 03 2016].

[5] POSTGREE, **Documentação POSTGREE** , [Online]. Available: http://www.postgreSQL.org/docs/9.5/static/index.html. [Acesso em 20 03 2016].

3

Indexação

3. Indexação

Uma das maiores qualidades de um Sistema Gerenciador de Banco de Dados (SGBD) é a rapidez com que as informações são resgatadas. Essa informação, geralmente é gravada na forma de arquivo em um disco rígido de computador. E em um SGBD, uma pesquisa feita entre milhares de *megabytes* pode ser retornada em questão de segundos. Na maioria das vezes não se tem a ideia da dimensão e complexidade que o Banco de Dados atinge em um sistema. Isso passa a ser um "mero" detalhe para quem o usa. O importante é que os dados estão sendo retornados rapidamente e a confiança nessa informação deve ser total. E para que essas duas qualidades sejam atendidas, rapidez e qualidade da informação, quem constrói o banco deve se preocupar com a organização dessa informação. Sem isso o retorno de dados pode ser demorado e o que é pior, pode não ter a veracidade que se espera de um banco de dados.

Uma das estruturas do Banco de Dados responsável pela rapidez nas respostas às pesquisas, são os índices. Eles organizam a informação de tal forma que fica fácil e rápido encontrá-la, sobretudo levando-se em conta os milhares de *megabytes* que um SBGD pode chegar. Um índice organiza os dados de uma tabela, fazendo com que a busca por esses dados seja feita com o mínimo de leituras, de maneira rápida e eficiente.

Esses índices são criados por meio de algoritmos eficientes de busca de dados, geralmente o algoritmo usado é o *B-Tree* ou, Árvore B.

Uma estrutura *B-Tree* possui uma "raiz" (primeiro nó da árvore, por onde sempre se iniciam as buscas), níveis intermediários e níveis folhas, conforme figura abaixo:

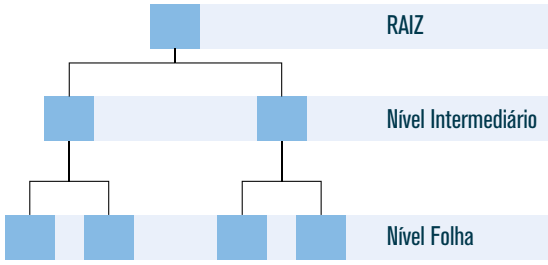


Figura 3.1 – B Tree. Fonte: Elaborada pelo autor.

Uma *B-Tree* sempre é simétrica, ou seja, possui os mesmos números de nós à esquerda e à direita de cada nível.

Cada nó, em um sistema de banco de dados é uma página de dados, para se ter uma ideia, o SQL Server da Microsoft, armazena até 8.060 bytes em uma página.

Logicamente, não é inteligente guardar todos os dados em um nó de uma árvore. A construção dessas estruturas é algo bastante complexa. Por isso cria-se essa estrutura apenas para uma parte da tabela, geralmente sua chave, a essa estrutura, dá-se o nome de ÍNDICE. Para as chaves de uma tabela, sempre são gerados índices. Mas por algum motivo de definição de projeto, ou quando se faz buscas constantes por determinados campos de uma tabela, pode ser interessante criar um índice específico para esses dados. É possível criar índices primários, secundários, clusterizados, não clusterizados, XML e especiais, entre outros.

OBJETIVOS

- Aprender sobre o significado e importância dos Índices;
- Conhecer e definir os tipos de Índices;
- Definir os Índices no SQL (Microsoft, Oracle, MySQL e PostgreSQL).

3.1 Tipos de índices

Os índices são estruturas facilitadoras de acessos a dados. Imagine o glossário de um livro, onde traz as principais palavras chaves em ordem alfabética ao final do livro. Isso facilita a busca de determinado assunto dentro do conteúdo do livro. Quando não existe essa estrutura no livro, o leitor é obrigado a fazer uma "varredura" em todas as páginas em busca da palavra ou assunto procurado. Quando existe o índice, todas as palavras chaves estão ordenadas e cada palavra diz em quais páginas elas são mencionadas. Um índice de nível único é basicamente um arquivo indexado. Já um índice de múltiplos níveis quebra esse arquivo em vários.

Existem basicamente os seguintes tipos de Índices:

- Índices de nível único:
  - Índices primários;
  - Índices clustering;
  - Índices secundários.
- Índices de múltiplos níveis;
- Índices densos;
- Índices não densos (esparso).

Um índice **primário** é um índice de nível único, pois é construído em apenas um arquivo e é criado mediante a escolha de uma chave primária de uma tabela, ou seja, escolhe-se um campo da tabela para ser o identificador único dessa tabela e sobre essa chave é criado o índice primário. Esse índice é responsável pela ordenação da chave primária e garante a unicidade dessa informação. Pelo fato do índice primário ordenar a chave primária de uma tabela e essa chave garantir a unicidade do registro na tabela, apenas um índice primário pode existir por tabela, consequentemente também existirá apenas uma chave primária por tabela.

O índice primário é criado em uma outra estrutura, outro arquivo ou tabela, com dois campos, o primeiro sendo a chave primária da tabela a ser indexada e o segundo campo, um ponteiro para um bloco de disco para o acesso a informação da tabela indexada.

O **índice clustering** é um índice de nível único, pois é construído em apenas um arquivo e é criado usando-se um campo que não é o campo chave, mas geralmente um campo onde sua informação é bastante acessada e deve ser ordenada para facilitar o acesso.

O **índice secundário** também é um índice de nível único, pois também é construído em apenas um arquivo e é criado para obter acesso a uma tabela para qual já existe um acesso primário. O índice secundário pode ser criado sobre um campo de valor único, mas não é uma chave primária ou sobre um campo que possui valores repetidos. Pode existir muitos índices secundários, tantos quantos forem necessários para o acesso a diversas colunas das tabelas de maneira mais rápida, mas deve-se levar em conta que a criação e manutenção dessas estruturas consomem processamento e recursos do Gerenciador de Banco de Dados.

Os **índices de múltiplos níveis** ao contrário dos de níveis únicos, são construídos mediante a criação de vários arquivos ou vários índices em níveis. Onde no primeiro nível a tabela é ordenada mediante a criação de um índice qualquer como visto anteriormente. Nos níveis seguintes, são criados índices primários sobre os índices anteriores. Pode-se construir índices multiníveis sobre qualquer índice, seja ele primário, clustering ou secundário. O problema de um índice de múltiplos níveis está quando é necessário excluir ou inserir um registro. O que antes estava distribuído em arquivos balanceados, acaba ficando desbalanceado. A operação para reorganizar esses índices se torna bastante complexa, já que a ordenação desses índices deve ser feita em tempo real, ou seja, exatamente no momento que a informação é manipulada (na inserção ou exclusão do registro).

O **índice denso** é constituído por um arquivo onde cada registro é indexado. Os registros podem estar armazenados em qualquer ordem do arquivo e seu arquivo de indexação consegue localizar cada registro em separado pelo arquivo de indexação.

Um **índice não denso** (também chamado de **índice esparso**) consiste em um arquivo de índice para blocos ou páginas do arquivo, cada um dos quais contendo um grupo de arquivo. Esse bloco precisa estar organizado segundo o atributo indexador. No arquivo de índice, cada índice aponta para o primeiro registro de cada bloco.

3.1.1 Índices Ordenados

Um índice ordenado é basicamente um guia ordenado para a busca de informação. Como o índice de um livro-texto, uma informação menor, ordenada que guiará para uma informação mais completa. Mas, pensando melhor, se toda a informação está no computador e com o computador podemos fazer o que queremos, por que não ordenar toda a informação? Isso não é possível?

A resposta é sim. É possível. E no começo foi assim, a informação era ordenada na mesma hora que era inserida. Mas a medida que a informação aumentava, a ordenação ficava mais difícil e o processo, mais lento. A ideia de "indexar" parte da informação para a busca agilizou todo o processo.

Um índice geralmente é definido sobre um campo importante de uma tabela de dados. Para esse campo é definido uma estrutura ordenada de acesso

à tabela. Ordenar o índice é mais rápido que ordenar a tabela inteira, e uma estrutura de ponteiros é definida ligando o índice ao campo da tabela, ficando mais rápido uma busca pelo índice, que além de estar ordenado é uma estrutura menor. De maneira geral, o índice armazena o valor de cada campo da tabela a ser ordenada e associa uma lista de ponteiros para todos os blocos de disco que contêm registros com aquele valor de campo. Um índice primário é criado sobre um campo chave da tabela a ser indexada. Se esse campo não é um campo chave, mas é um campo onde faz-se necessário a criação de uma estrutura para busca, esse índice é chamado índice clustering. Um índice pode ter, no máximo, um campo de classificação física, portanto, um índice ou é primário ou é clustering, mas não ambos. Um terceiro índice de classificação pode ser especificado sobre qualquer outro campo da tabela que não seja o campo de classificação da tabela, esse índice é chamado índice secundário e uma tabela pode ter vários índices secundários além do índice de classificação primária.

3.1.2 Índices Densos e Esparsos

Os índices também podem ser classificados como densos ou esparsos. Um índice denso possui uma entrada de índice para cada valor da chave de busca (portanto, para cada ocorrência do valor no registro) do arquivo de dados. Porém, um índice esparso (ou não-denso) possui entradas de índice para apenas alguns dos valores de busca. Um índice primário é, portanto, um índice esparso, uma vez que inclui uma entrada de índice para cada bloco do arquivo de dados e as chaves de seus registros âncoras em vez de uma entrada para cada valor de busca (ou cada registro). O arquivo de índice para um índice primário precisa de muito menos blocos do que o arquivo de dados por duas razões. Primeiro, há menos entradas de índice que registros no arquivo de dados. Segundo, cada entrada de índice é tipicamente menor em tamanho que o registro de dados, porque elas possuem apenas dois campos; consequentemente, mais entradas de índice do que registros de dados podem caber em um bloco. Portanto, uma busca binária no arquivo de índice exige menos acessos a blocos do que uma busca binária no arquivo de dados.

Um dos primeiros métodos de indexação criado utiliza um arquivo indexado sequencialmente, que são utilizados tanto para um acesso sequencial como para um acesso aleatório aos registros. Exemplificando, podemos criar um arquivo de índices para os departamentos existentes em uma empresa.

Para localizar os funcionários pelos departamentos mais rapidamente, podemos criar um arquivo de índice onde todos os departamentos são ordenados sequencialmente e "apontar" cada departamento para os departamentos em que cada funcionário está alocado:

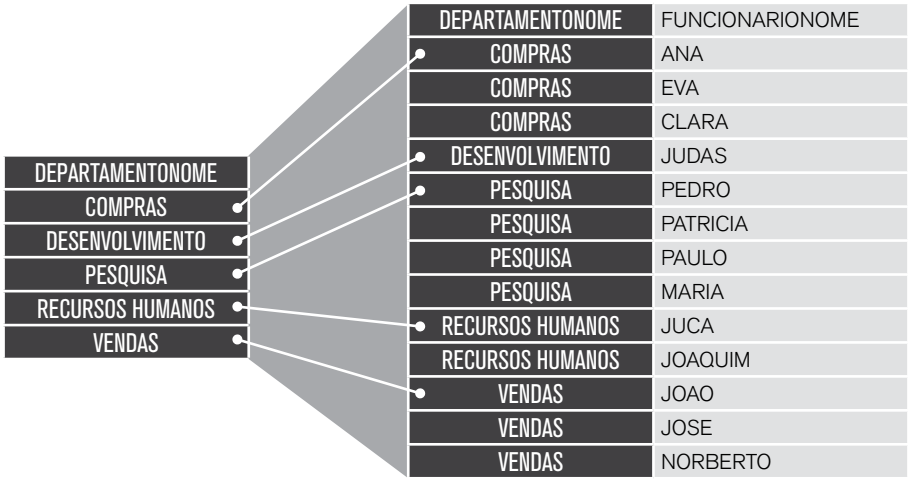


Figura 3.2 – Índice de Departamentos. Fonte: Elaborada pelo autor.

Reparem que no exemplo, para cada departamento da empresa existe um registro de índice, mesmo que exista apenas um departamento para um determinado funcionário. Este tipo de índice é chamado de índice denso, pois ele irá conter todos os nomes possíveis de indexação para uma tabela. Existem também os índices "esparsos", neles ao invés de criar todas as entradas existentes de valores, apenas alguns valores são criados. Para localizar um registro, encontramos a entrada de índice com o maior valor de chave de procura que seja menor ou igual ao valor de chave de procura que estamos procurando. Iniciamos a busca no registro apontado para a tabela de índices e seguimos para a outra tabela até encontrarmos o registro desejado. Por exemplo, se criássemos um arquivo de índice esparso sem o valor "RECURSOS HUMANOS" na tabela de índices para o Nome de departamentos, a busca na tabela de índices encontraria o valor mais próximo "PESQUISA", iria para o primeiro registro equivalente na tabela de funcionários e prosseguiria a busca até encontrar o Departamento RECURSOS HUMANOS:

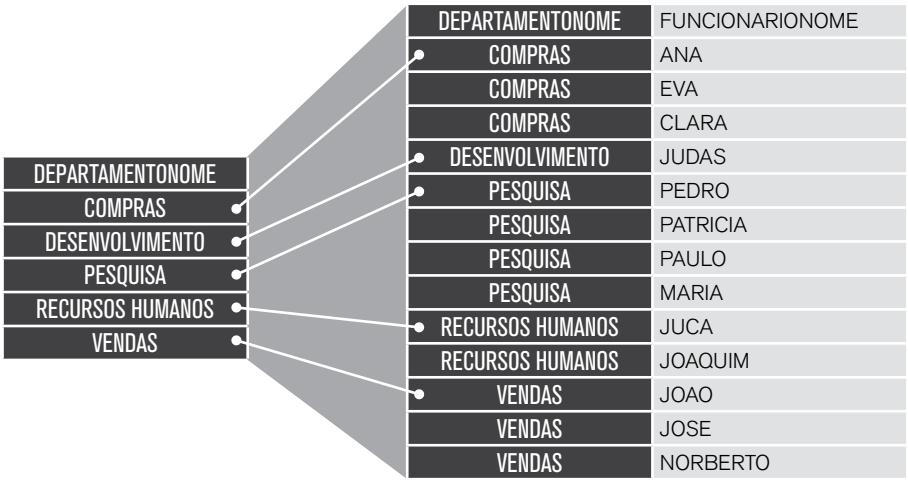


Figura 3.3 – Índice Esparso. Fonte: Elaborada pelo autor.

Em termos de eficiência de buscas, que na maioria é o que se deseja de um sistema de banco de dados, podemos notar que um índice denso é muito mais eficiente e dessa forma preferível de ser implementado. A vantagem de um índice esparso é o fato de ocupar pouco espaço em disco e também menos trabalho na manutenção no que se refere a inclusões e exclusões de registros de índice.

3.1.3 Índice Cluster e Não Cluster

Um índice cluster determina a ordem em que as linhas de uma tabela (ou entidade) são armazenadas no disco. Se uma tabela tem um índice cluster, no momento de uma inserção de dados os registros dessa tabela serão armazenados em disco na mesma ordem do índice. Por exemplo, suponha que temos uma tabela chamada "DEPARTAMENTO" que tem uma coluna de chave primária "DEPARTAMENTO_ID" e que criamos um índice cluster para essa mesma coluna. Ao fazer isso, todas as linhas dentro da tabela DEPARTAMENTO serão fisicamente ordenadas (no disco atual em que estão inseridas) através dos valores que estão na coluna DEPARTAMENTO_ID. Pelo fato de um índice cluster estar associado a uma chave primária, onde só existe uma por tabela, também somente um índice cluster é permitido por tabela.

Isso implicará em um ganho enorme na performance das pesquisas, pois as colunas da tabela (ou entidade) estarão ordenadas na mesma ordem dos

índices clusterizados por intermédio do modelo de armazenamento usado por esse tipo de índice. Já os índices não-cluster não fazem esse trabalho de ordenação dos dados tal qual é feito com os índices clusterizados. Ou seja, enquanto os índices clusterizados ordenam fisicamente tanto as linhas da tabela (ou entidade) quanto os próprios índices e mantêm os mesmos próximos uns aos outros; os não-clusterizados ordenam somente o índice em si, e não as linhas (que são salvas de forma aleatória no disco).

Numa tabela, quando é definida uma chave primária, um índice cluster é criado automaticamente pelo banco para essa tabela. E o que ele faz basicamente é ordenar as informações pela coluna de chave daquele índice em disco.

Com a criação de um índice cluster em uma tabela (ou entidade), os dados dessa tabela seriam sempre ordenados fisicamente seguindo a ordenação do campo chave. E sempre que um novo dado de registro fosse inserido ou atualizado, toda a tabela seria reorganizada para manter a ordenação correta. Essa abordagem faz mais sentido com dados que não se alteram com frequência, por exemplo, uma tabela de CEPs. Onde os CEPs raramente mudam ou são criados. Os índices não clusterizados, já fazem sentido onde as informações são constantemente atualizadas ou inseridas. E pelo fato de um índice não clusterizado não estar associado a apenas um campo (chave primária) ele pode ser criado para cada coluna que se deseja ordenar. Os índices não cluster são muito usados quando necessitamos encontrar uma informação de várias formas diferentes como por exemplo, queremos encontrar um livro pelos campos Nome, Autor, Editora, Assunto etc.

3.1.4 Índice Multinível

Um índice Multinível pode ser definido como um índice de índices, são construídos mediante a criação de vários índices em níveis. No primeiro nível é armazenado um "*range*" de valores e um ponteiro para cada chave em um segundo nível, esse segundo nível possui uma parte das chaves e um novo ponteiro para o acesso à informação propriamente dita. Isso facilita a busca já que é necessária uma quantidade menor de iterações de busca quando os índices são quebrados em níveis. Esse tipo de índice se mostra mais complexo, principalmente para a manutenção de entrada e exclusão de dados, já que ele precisa ficar organizado em diversos arquivos mantendo os "*ranges*" de valores balanceados. Uma solução pode ser deixar espaços previamente alocados para efetuar as inclusões e exclusões, para não ter que ordenar todos os arquivos de índices novamente.

Exemplo de uma estrutura de índices em dois níveis:

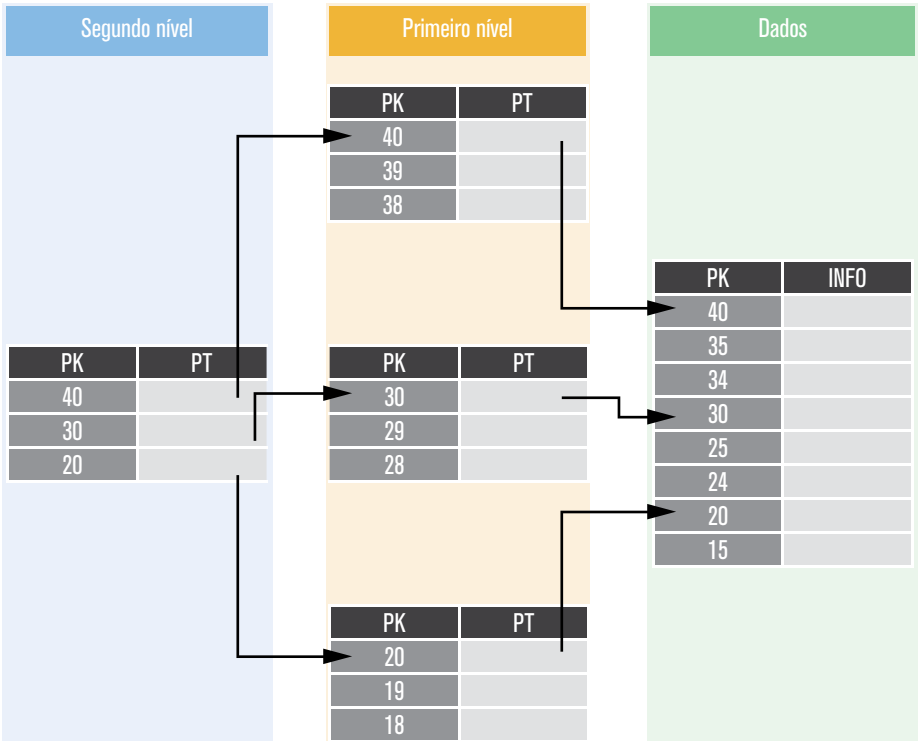


Figura 3.4 – Estrutura Multinível. Fonte: Elaborada pelo autor.

3.2 Definição de índice em SQL

Cada Servidor de Banco de Dados tem suas particularidades na criação de índices. Embora os tipos básicos são praticamente iguais entre os fabricantes, algumas peculiaridades podem existir.

Veja a seguir algumas tabelas onde são mostrados os vários índices implementados nas várias versões do SQL.

No SQL da Microsoft temos os seguintes índices:

TIPO DE ÍNDICE	DESCRIÇÃO
CLUSTERED	Um índice cluster armazena as linhas de dados da tabela ou exibição em ordem com base na chave de índice de cluster. O índice de cluster é implementado como uma estrutura de índice B-tree que suporta a recuperação rápida das linhas, com base em seus valores de chave de índice de cluster.
NONCLUSTERED	Um índice não cluster pode ser definido em uma tabela ou exibição com um índice de cluster ou em um heap. Cada linha de índice no índice não cluster contém o valor de chave não cluster e um localizador de linha. Este localizador aponta para a linha de dados no índice clusterizado ou heap tendo o valor da chave. As linhas no índice são armazenadas na ordem dos valores de índice de chave, mas as linhas de dados não são garantidas para estar em qualquer ordem particular, a menos que um índice de cluster é criado na tabela.
UNIQUE	Um índice *Unique* garante que a chave de índice não contém valores duplicados e, portanto, todas as linhas na tabela ou exibição é, de alguma forma única.
	Uniqueness pode ser uma propriedade de ambos os índices clusterizados e não clusterizados.
COLUMNSTORE	Um índice *columnstore* é criado com base em particionamento vertical dos dados por colunas, armazenados como objetos grandes (LOB).
ÍNDICE COM COLUNAS INCLUÍDAS	Um índice agrupado que é estendido para incluir colunas não-chave para além das colunas de chave.
ÍNDICE EM COLUNAS COMPUTADAS	Um índice em uma coluna que é derivado a partir do valor de uma ou mais colunas, ou certas entradas determinísticas.
FILTERED	Um índice não clusterizado otimizado, especialmente adequado para cobrir consultas que selecionar a partir de um subconjunto bem definido de dados. Ele usa um predicado de filtro para indexar uma porção de linhas da tabela. Um índice *Filtered* pode melhorar o desempenho da consulta, reduzir os custos de manutenção do índice, e reduzir os custos de armazenamento comparado com os índices de tabela completa.
SPATIAL	Um índice *Spatial* fornece a capacidade de executar determinadas operações de forma mais eficiente em objetos espaciais *(dados espaciais)* em uma coluna do tipo de dados de **geometria.** O índice espacial reduz o número de objetos sobre os quais operações espaciais relativamente dispendiosas precisam ser aplicadas.
XML	A desfiado, e persistiu, a representação de XML objetos binários grandes (BLOBs) na coluna tipo **de** dados **XML.**
FULL-TEXT	Um tipo especial de índice funcional baseado no token que é construído e mantido pelo *Microsoft Full-Text Engine* para o SQL Server.Ele fornece suporte eficiente para a palavra pesquisas sofisticadas em dados de cadeia de caracteres.

Figura 3.5 – Tipos de Índices no MSSQL. Fonte: Elaborada pelo autor.

No SQL Oracle temos:

TIPO DE ÍNDICE	DESCRIÇÃO
NORMAL INDEX	Índice padrão. Por default o Oracle database cria índices B-Tree.
BITMAP INDEX	Usados para colunas com baixa cardinalidade, ou seja, com baixa variação de valores (por exemplo uma coluna GENERO onde os valores temos apenas F ou M, para masculino ou feminino.
PARTITIONED INDEX	Índice criado em vários blocos para atender as tabelas particionadas ou segmentadas disponíveis no ORACLE.
INDEX BASEADO EM FUNÇÃO	É um índice baseado no valor de retorno de uma expressão ou função.
DOMAIN INDEXES	Usando o objeto de esquema Index Type, um índice específico do aplicativo pode ser criado. Esse índice é chamado de índice de domínio, uma vez que é utilizada para a indexação de dados em domínios específicos do aplicativo.

Figura 3.6 – Tipos de Índices no oracle. Fonte: Elaborada pelo autor.

No MySQL:

TIPO DE ÍNDICE	DESCRIÇÃO
UNIQUE	Um índice *Unique* garante que a chave de índice não contém valores duplicados e, portanto, todas as linhas na tabela ou exibição é, de alguma forma única. *Uniqueness* pode ser uma propriedade de ambos os índices clusterizados e não clusterizados.
SPATIAL	Um índice *Spatial* fornece a capacidade de executar determinadas operações de forma mais eficiente em objetos espaciais *(dados espaciais)* em uma coluna do tipo de dados de **geometria.** O índice espacial reduz o número de objetos sobre os quais operações espaciais relativamente dispendiosas precisam ser aplicadas.
FULL-TEXT	Um tipo especial de índice funcional baseado no token que é construído e mantido pelo *Microsoft Full-Text Engine* para o SQL Server. Ele fornece suporte eficiente para a palavra pesquisas sofisticadas em dados de cadeia de caracteres.

Figura 3.7 – Tipos de Índices no MySQL. Fonte: Elaborada pelo autor.

ATIVIDADES

01. Em sua opinião, qual a importância na criação de índices em um sistema de banco de dados?

02. Defina índice primário.

03. Defina índice secundário.

04. Por que apenas um índice primário pode existir?

05. Quais são as diferenças entre índices densos e índices esparsos?

REFERÊNCIAS BIBLIOGRÁFICAS

[1] TENEMBAUM, Aaron - **Estruturas de Dados Usando C** - Ed. Person Makron Books – 2004.

[2] ELMASRI, R.; NAVATHE, S., **Sistemas de Banco de Dados**. Pearson Education do Brasil, 4ª.Ed 2005.

[3] PINHEIRO, Álvaro Farias, **Fundamentos de Engenharia de Software**: Introdução a Banco de dados, Volume II, 5ª Ed 2015.

[4] Microsoft SQL SERVER. **Documentação Microsoft SQL Server 2014**, [Online]. Available: https://msdn.microsoft.com/library/mt590198.aspx. [Acesso em 20 03 2016].

[5] ORACLE SERVER, **Documentação ORACLE**, [Online]. Available: http://www.oracle.com/technetwork/pt/indexes/documentation/index.html. [Acesso em 20 03 2016].

[6] MYSQL, **Documentação MYSQL**, [Online]. Available: https://dev.mySQL.com/doc/refman/5.7/en/. [Acesso em 20 03 2016].

[7] POSTGREE, **Documentação POSTGREE** , [Online]. Available: http://www.postgreSQL.org/docs/9.5/static/index.html. [Acesso em 20 03 2016].

4

Transações

4. Transações

Poder guardar milhares e milhares de informação e fazê-la de forma organizada é uma das grandes vantagens de um Sistema de Banco de Dados (SGBD). Mas como saber se toda essa informação é realmente confiável? Como saber se todas as contas que um banco de dados é capaz de fazer, estão sendo feitas de maneira correta, sendo arquivada junto a alguma informação que faça sentido? Imagine se todas as transações bancárias não pudessem ser confiáveis? Se ao criar um novo registro, ao tentar recuperá-lo descobríssemos que a informação que deveria estar lá está fazendo parte de outro registro? Todas essas respostas são dadas pelo sistema de "transações" de um Banco de Dados. É com ele que garantimos que um registro de banco de dados está sendo guardado no lugar certo, junto com as informações certas. Ele é capaz de "guiar" toda a forma que as informações são salvas.

Sistemas de transação são sistemas desenvolvidos para bancos de dados que garantem a execução de processamento de registros concorrentes no banco de dados. Neste capítulo apresentaremos os conceitos de processamentos de transações.

Veremos como uma transação deve ser completa e integral para garantir precisão. Analisaremos o problema das "concorrências" que surgem quando diversas transações, submetidas por vários usuários, interferirem nas outras, produzindo resultados imprecisos. Veremos também como é possível recuperar uma informação, caso uma transação não seja feito da maneira esperada.

OBJETIVOS

- Definir e conhecer o significado de Transacoes em Bancos de dados;
- Conhecer os estados das Transações;
- Definir as propriedades ACID.

4.1 Conceito de Transação

Um dos conceitos mais simples e que define muito bem o conceito de transação é o dado por "ELMASRI, R.; NAVATHE, S., *Sistemas de Banco de Dados – pag. 402*": "*Uma transação é uma unidade atômica de trabalho que ou estará completa ou não foi realizada*". Simples assim, uma transação é algo que dá a garantia de ter ocorrido de maneira satisfatória e completa.

Uma transação inclui uma ou mais operações de acesso ao banco de dados. Podem incluir operações de inserção, alteração, exclusão ou recuperação. Uma operação de transação pode estar definida em um programa de aplicação ou pode ser definida e executada mediante instruções especificas da linguagem de banco de dados, tal como o SQL. A linguagem SQL, por meio de comandos de início e fim podem estabelecer explicitamente as operações a serem executadas garantindo-se uma transação. Todos os comandos de acesso a dados presentes entre esse início e fim são considerados parte da transação.

4.2 Estados da Transação

Como dito anteriormente: "*Uma transação é uma unidade atômica de trabalho que ou estará completa ou não foi realizada*". Portanto, para controlar uma transação, poder definir como completa e realizada, ou até mesmo para poder retorna-la ao estado que a originou, o sistema precisa manter o controle de quando a transação inicia, termina e de suas efetivações ou cancelamentos (*commits* ou *aborts*). Dessa forma, deve-se ter o controle das seguintes operações:

- BEGIN_*TRANSACTION*: Marca o início da execução da transação.
- READ ou *WRITE*: Especificam operações de leitura ou gravação em itens do banco de dados, que são executadas como parte de uma transação.
- END_*TRANSACTION*: Especifica que as operações *READ* e *WRITE* da transação terminaram, e marca o fim da execução da transação. Entretanto, nesse ponto é necessário verificar se as mudanças introduzidas pela transação podem ser permanentemente aplicadas ao banco de dados (efetivadas), ou se a transação deverá ser.
- COMMIT_*TRANSACTION*: Indica término com sucesso da transação, de forma que quaisquer alterações (atualizações) executadas poderão ser seguramente efetivadas no banco de dados e não serão desfeitas.

• ROLLBACK (ou *ABORT*): Indica que uma transação não terminou com sucesso, de forma que quaisquer mudanças ou efeitos que a transação possa ter aplicado ao banco de dados deverão ser desfeitas. Uma transação entra em estado ativo imediatamente após o início de sua execução, no qual poderá emitir operações *READ* (leitura) e *WRITE* (gravação). Quando a transação termina, ela passa para o estado de efetivação parcial. Nesse ponto, alguns protocolos de restauração precisam garantir que uma falha de sistema não impossibilite a gravação permanente das mudanças promovidas pela transação (geralmente pela gravação de mudanças no log do sistema, que será visto na próxima seção). Uma vez atendidas todas as verificações, diz-se que a transação alcançou seu ponto de efetivação, e entra, então, no estado de efetivação (*committed state*). Uma vez efetivada, a transação tem sua execução concluída com sucesso, todas as suas mudanças serão gravadas permanentemente no banco de dados.

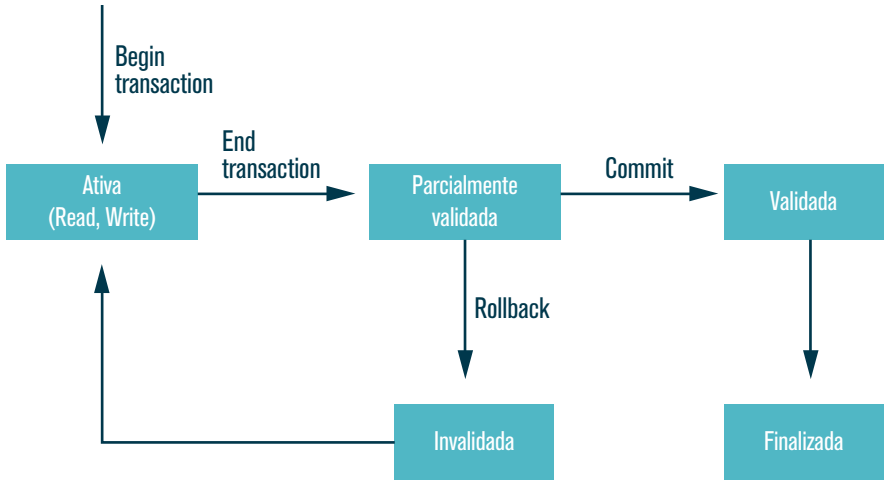


Figura 4.1 – Diagrama de estados. Fonte: Elaborada pelo autor.

4.3 Propriedades ACID

ACID (acrônimo de Atomicidade, Consistência, Isolamento e Durabilidade - do inglês: *Atomicity, Consistency, Isolation, Durability*), é um conceito utilizado em ciência da computação para caracterizar uma transação em um Banco de Dados, ou seja, para garantir uma transação ele deve ser "**Atômica**", no sentido de única, sem ficar nada pendente. Deve ser "**Consistente**", deve respeitar as

regras de integridade dos dados, suas chaves e tipos de dados. Deve ser "**Isolada**", ou seja, o trabalho naquele registro, pertinente a aquela transação deve ser único, sem a interferência de nenhuma outra transação ou operação que possa estar ocorrendo. Deve ser "**Durável**", uma vez executada e finalizada, seus dados devem ser persistentes independente de qualquer ocorrência externa, tais como uma queda de energia ou interrupção abrupta do sistema. Uma vez finalizada seu resultado será confiável.

EXEMPLO

- **Atomicidade**

Ou todo o trabalho é feito, ou nada é feito.

Em uma transferência de valores entre contas bancárias, é necessário que, da conta origem seja retirado um valor X e na conta destino seja somado o mesmo valor X. As duas operações devem ser completadas sem que qualquer erro aconteça, caso contrário todas as alterações feitas nessa operação de transferência devem ser desfeitas;

- **Consistência**

Deve respeitar as regras de chaves e tipos de dados.

Considere um banco de dados que guarde informações de produtos e que use o Código de Barras como chave primária, no formato de 11 dígitos numéricos. Então, qualquer inserção ou alteração no banco não pode duplicar um código de barras (unicidade de chaves) ou colocar um valor de código de barras inválido, como o valor 12356789AB (restrição de integridade lógica).

- **Isolamento**

Evita que transações paralelas interfiram umas nas outras.

Considere as duas seguintes transações efetuadas em um banco de dados com informações de funcionários de uma empresa (o exemplo a seguir é baseado no exemplo disponível no wikipedia. Disponível em: <https://pt.wikipedia.org/wiki/ACID>. Acessado em 22 abr. 2016):

T_1: Aumentar o preço de um produto em 10 reais para o produto de código de barras 12345678901.

T_2: Aumentar o preço de um produto em 10% para o produto de código de barras 12345678901.

Podemos visualizar as transações da seguinte forma:

T_1:

seleciona produto que tenha Código de Barras igual à 12345678901;

altera o valor somando 100 reais;

faz o *commit* (confirma a gravação no banco de dados).

T_2:

seleciona funcionário que tenha Código de Barras igual à 12345678901;

altera o valor somando 10%;

faz o *commit* (confirma a gravação no banco de dados).

Se as duas transações forem executadas sequencialmente, o preço final do produto deve ser (o produto tem valor inicial de 1000 reais):

T_1 e depois T_2: valor passa a ser 1100 reais e depois, somando 10% de 1100, passa a ser 1210 reais.

T_2 e depois T_1: valor passa a ser 1100 reais e depois, somando 100 reais, passa a ser 1200 reais.

Então, se executarmos estas duas transações em paralelo, o valor final do preço desse produto deve ser de 1200 ou de 1210 reais.

Se as duas transações não estiverem isoladas (uma puder ver os resultados parciais da outra), então, quando executadas em paralelo, pode ocorrer o seguinte:

T_1 seleciona o produto;

T_2 seleciona o mesmo produto;

T_1 altera o preço para 1100 reais;

T_2 altera também o preço para 1100 reais;

T_1 grava a mudança no banco de dados;

T_2 também grava a mudança no banco de dados.

Assim, no fim, o preço deste produto será igual à 1100 reais (valor gravado por T_2). Isto ocorreu neste exemplo porque deixamos as duas transações alterarem o mesmo campo (o preço) ao mesmo tempo. A propriedade de isolamento não permite que isto ocorra.

- **Durabilidade**

Os efeitos de uma transação em caso de sucesso (*commit*) devem persistir no banco de dados mesmo em casos de quedas de energia, travamentos ou erros. Garante que os dados estarão disponíveis em definitivo.

4.4 Execução Concorrente de Transações

Vimos nos itens anteriores que uma transação ocorre mediante alguns cuidados que devem ser tomados pelo sistema de banco de dados. Esses passos garantem que as propriedades "ACID" são presentes em uma transação. Além disso, essas transações ocorrem a todo tempo em um sistema de banco de dados. Mesmo as operações simples de leitura e escrita estão correndo a todo momento no sistema. Quando transações manipulam dados concorrentemente, alguns problemas podem ocorrer tais como acesso inconsistente a dados, perdas de atualização e perda da consistência do banco.

A maioria dessas técnicas assegura a serialização de planos de execução, ou seja, garante que as instruções são executadas uma após a outra, sem execuções simultâneas, usando algumas regras que garantem a serialização. Um importante conjunto de protocolos emprega a técnica de bloqueio dos itens de dados para impedir que múltiplas transações acessem os itens concorrentemente. Protocolos de bloqueio são usados na maioria dos Sistemas de Bancos de Dados comerciais. Um outro conjunto de protocolos de controle de concorrência usa *timestamps* (marcas de tempo). Um *timestamp* é um identificador único para cada transação, gerado pelo sistema. Outro protocolo de bloqueio é conhecido como protocolo de controle de concorrência de multiversão, que usam múltiplas versões de um item de dado. Temos também um protocolo baseado no conceito de validação ou certificação de uma transação depois que ela executa suas operações; estes são, às vezes, chamados protocolos otimistas. Um outro fator que afeta o controle de concorrência é a granularidade dos itens de dados — isto é, qual porção do banco de dados um item de dado representa. Um item pode ser tão pequeno quanto o valor de um único atributo (campo) ou tão grande quanto um bloco de disco, ou todo um arquivo, ou um banco de dados inteiro.

As técnicas de bloqueio usadas para o controle de concorrência são baseadas no conceito de bloqueio de itens de dados, ou LOCK. Um bloqueio (LOCK) é um controle associado a um item de dados que descreve a condição do item em relação às possíveis operações que podem ser aplicadas a esse dado.

4.5 Bloqueios no mecanismo de banco de dados

Também conhecido como LOCK, um bloqueio é uma técnica bastante útil no controle das alterações de um sistema de banco de dados. Geralmente há um

bloqueio para cada item de dado no banco de dados e eles são usados como meio de sincronizar o acesso por transações concorrentes aos itens do banco de dados. Ao contrário de uma transação, um bloqueio não garante um processo de manipulação de dados de maneira consistente. Um bloqueio apenas impede que outros recursos acessem a tabela em LOCK. Por esse motivo o uso deliberado do LOCK deve ser ponderado pois podem ocorrer DEADLOCKS (impasse) ou STARVATION (inanição). Um *deadlock* ocorre quando um objeto bloqueado está esperando algum outro recurso que também está em bloqueio e, por conseguinte, esse mesmo processo está esperando o primeiro que também está na condição de bloqueio. Já a condição de *starvation* ou inanição ocorre quando um processo em bloqueio não pode continuar por um período indefinido de tempo, enquanto outras transações no sistema continuam normalmente.

4.6 Controle de Transação em SQL (*Commit, Rollback, Savepoint*)

Vimos que uma transação é uma ocorrência ordenadas de escritas e leituras em um banco de dados com um resultado único e consistente de dados preservando a integridade dos mesmos. Esse processamento pode ser executado todo ou não garantindo a atomicidade das informações.

A sintaxe básica de uma transação é:

Begin Transaction

< comandos >

Commit ou Rollback

Onde:

Begin Transaction: *Tag* inicial para o início de uma transação.

< Comando >: Conjunto de comando a serem executados dentro de uma transação.

Commit ou *Rollback*: Comandos que finalizam a transação onde o *commit* confirma o conjunto de comandos e o *rollback* desfaz toda a operação, executado pelo corpo de comandos caso tenha ocorrido algum evento não desejado.

Verificando erros dentro de uma transação:

No SQL SERVER temos uma varável de sistema que faz a identificação de um erro dentro de uma transação chamada de '@@ERROR' variável essa que por padrão recebe o valor 0 (zero) caso não ocorra nem um erro, e recebe o valor 1 (um) caso algum erro ocorra.

EXEMPLO

```
@@ERROR:

BEGIN TRANSACTION

UPDATE FUNCIONARIO
SET SALARIO = SALARIO * 1.10
WHERE SALARIO = 1000
IF @@ERROR = 0
      COMMIT
ELSE
      ROLLBACK
END
```

Nesse exemplo estamos dando um aumento de salário de 10% para todos os funcionários cujo salário é mil reais. Caso algum erro ocorra, o comando é revertido, caso não tenha erro, toda operação é validada.

Se ocorrer algum erro, uma mensagem do sistema será apresentada, às vezes essas mensagens são um tanto complicadas e geralmente apresentadas na linguagem de instalação do banco de dados, dessa forma, podemos fazer o uso de uma função no SQL responsável por "gerar" um erro com as características definidas pelo administrador o programador da rotina.

Ficando assim:

```
BEGIN TRANSACTION

UPDATE FUNCIONARIO
SET SALARIO = SALARIO * 1.10
WHERE SALARIO = 1000
IF @@ERROR = 0
      COMMIT
ELSE
      RAISERROR(5001,16,1,'ERRO AO ALTERAR SALARIO, TABELA FUNCIONARIO!')
      ROLLBACK
END
```

Nessa melhoria de código, ao ocorrer algum problema na atualização do salário uma mensagem amigável de erro será apresentada.

Nem sempre, um *rollback* em execuções serão feitos somente com a ocorrência de erro. Podemos ter uma situação lógica em uma transação que provoque um *rollback*. Dessa forma uma RAISEERROR pode intimidar o usuário, fazendo com que ele ache que ocorreu um erro "catastrófico". Para isso, podemos fazer o uso de uma propriedade PRINT, ou até mesmo apresentar um SELECT amigável:

```
BEGIN TRANSACTION

UPDATE FUNCIONARIO
SET SALARIO = SALARIO * 1.10
WHERE SALARIO = 1000
IF @@ERROR = 0
      COMMIT
ELSE
      PRINT 'NAO FOI POSSIVEL ALTERAR O SALARIO, TABELA FUNCIONARIO!'
      ROLLBACK
END
```

Essa transação é dita 'explicita', pois requer que o comando BEGIN TRANSACTION seja executado para termos uma transação em andamento. Esse é o modo padrão com que o Microsoft SQL Server é executado. Mas, podemos fazer com que todas as operações sejam executadas mediante as regras de transação. Para isso basta executar o comando abaixo uma vez, e assim, todas as execuções desse ponto em diante serão tratadas como sendo uma transação. Perceba que não mais será necessário a execução de um begin transaction, mas o programador do banco de dados deverá executar o COMMIT ou ROLLBACK quando necessário.

```
SET IMPLICIT_TRANSACTIONS ON

UPDATE FUNCIONARIO
SET SALARIO = SALARIO * 1.10
WHERE SALARIO = 1000
IF @@ERROR = 0
      COMMIT
ELSE
```

```
        PRINT 'NAO FOI POSSIVEL ALTERAR O SALARIO, TABELA FUNCIONARIO!'
        ROLLBACK
    END
```

Pode-se ainda fazer o uso do comando a seguir:

```
SET XACT_ABORT ON
```

O padrão dessa propriedade é OFF. Colocando em ON o sistema de banco automaticamente executará um ROLLBACK caso algum erro ocorra.

Outra característica interessante de um banco de dados é o uso.

Suponhamos que em nosso sistema tenhamos várias execuções de INSERT e UPDADE e que por algum motivo, queiramos controlar as transações de forma que elas possam ser executadas e que sob determinados aspectos somente parte dessa transação se torne válida, sem que precisemos fazer um ROLLBACK em toda a transação e comecemos novamente para fazer somente a parte que nos interesse. Para isso pode-se recorrer a uma propriedade interessante, chamada SAVE POINT. Com ela pode-se definir alguns pontos da transação onde será possível o ROLLBACK ou COMMIT apenas daquela parte, exemplificando:

```
BEGIN;
   CREATE TEMPORARY TABLE tabela1 (col1 int) ON COMMIT DROP;
   INSERT INTO tabela1 VALUES (1);
   SAVE TRANSACTION meu_ponto_de_salvamento;
   INSERT INTO tabela1 VALUES (2);
   ROLLBACK TO SAVEPOINT meu_ponto_de_salvamento;
   INSERT INTO tabela1 VALUES (3);
   SELECT * FROM tabela1;
COMMIT;
```

A transação acima inseriu os valores 1 e 3, mas não o 2.

Nos sistemas de bancos de dados POSTGRE, ORACLE e MYSQL ao invés de escrever SAVE TRANSACTION é escrito SAVEPOINT:

```
BEGIN;
   CREATE TEMPORARY TABLE tabela1 (col1 int) ON COMMIT DROP;
   INSERT INTO tabela1 VALUES (1);
   SAVEPOINT meu_ponto_de_salvamento;
   INSERT INTO tabela1 VALUES (2);
   ROLLBACK TO SAVEPOINT meu_ponto_de_salvamento;
```

```
    INSERT INTO tabela1 VALUES (3);
    SELECT * FROM tabela1;
  COMMIT;
```

Ainda, no sistema de banco de dados do POSTGRE e MYSQL (no ORACLE não) existe um comando caso o programador queira liberar um SAVE POINT. O comando é o RELEASE SAVEPOINT esse comando irá destruir o SAVEPOINT previamente criado:

```
BEGIN;
  CREATE TEMPORARY TABLE tabela1 (col1 int) ON COMMIT DROP;
  INSERT INTO tabela1 VALUES (3);
  SAVEPOINT meu_ponto_de_salvamento;
  INSERT INTO tabela1 VALUES (4);
  RELEASE SAVEPOINT meu_ponto_de_salvamento;
  SELECT * FROM tabela1;
COMMIT;

 col1
------
    3
    4
(2 linhas)
```

A transação acima inseriu tanto o 3 quanto o 4.

ATIVIDADES

01. O que é execução concorrente de transações em banco de dados num sistema multiusuário? Em sua opinião, por que o controle de concorrência é necessário?

02. O controle de transação no exemplo abaixo é necessário, por quê?

```
begin tran
select * from PESSOAS
commit tran
```

03. O controle de transação no exemplo abaixo é necessário, por quê?

```
begin tran
update PESSOAS set nome = 'ERNERSTINO' where CPF = '665.854.455-70'
commit tran
```

04. Considere:

I. Se uma transação é concluída com sucesso (operação commit bem sucedida), então seus efeitos são persistentes.

II. Ou todas as ações da transação acontecem, ou nenhuma delas acontece.

As propriedades (I) e (II) das transações em SGBDs, significam, respectivamente:

a) Durabilidade e consistência.
b) Persistência e automação.
c) Isolação e atomicidade.
d) Durabilidade e atomicidade.
e) Consistência e persistência.

05. É possível dividir uma transação em etapas? Dê exemplo.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] TENEMBAUM, Aaron - **Estruturas de Dados Usando C** - Ed. Person Makron Books – 2004.

[2] ELMASRI, R.; NAVATHE, S., **Sistemas de Banco de Dados**. Pearson Education do Brasil, 4ª.Ed 2005.

[3] PINHEIRO, Álvaro Farias, **Fundamentos de Engenharia de Software**: Introdução a Banco de dados, Volume II, 5ª Ed 2015.

[4] **Definicção das propriedades ACID**, Wikipédia. Disponivel em: https://pt.wikipedia.org/wiki/ACID. Acessado em: 22/04/2016.

5

Otimização e Processamento de Consultas

5. Otimização e Processamento de Consultas

Neste capítulo veremos as técnicas utilizadas por um SGBD para processar, otimizar e executar consultas de alto nível. As pesquisas feitas usando-se linguagem SQL, ou QUERYs, em geral são traduzidas em uma linguagem da álgebra relacional e otimizadas para terem um tempo de processamento aceitável quando executadas.

Essas etapas de tradução e otimização são feitas pelo gerenciador de banco de dados. Encontrar um equilíbrio entre elas é uma tarefa que requer bastante processamento e algoritmos eficientes.

Existem diferentes algoritmos responsáveis pela otimização e tradução para implementar o acesso eficiente aos dados de um gerenciador de banco de dados (SGBD). Esses algoritmos criam um plano de execução levando-se em conta não somente a melhor estratégia, mas devem procurar a estratégia mais eficiente.

As consultas escritas na linguagem SQL são lidas pelo gerenciador de banco de dados (SGBD), analisadas e validadas. São usadas técnicas conhecidas da teoria de compiladores passando por análises e sínteses. A consulta é analisada sintaticamente então é validada e depois traduzida para uma expressão em álgebra relacional e logo em seguida é otimizada pelo otimizador de consultas. O algoritmo usado pelo SGBD busca essencialmente a tradução da consulta escrita em SQL para um código otimizado onde o custo de acesso aos dados será o menor possível.

OBJETIVOS

- Conhecer alguns dos algoritmos para processamento de consultas;
- Aprender a otimizar uma consulta;
- Conhecer um plano de execução de consultas.

5.1 Algoritmos para processamento de consultas

"Uma consulta expressa em uma linguagem de consulta de alto nível, tal como SQL, deve primeiro passar por uma análise léxica, uma análise sintática e ser validada". "ELMASRI, R.; NAVATHE, S., *Sistemas de Banco de Dados – pag. 354*".

A análise léxica identifica as palavras-chave da linguagem SQL, enquanto que a análise sintática identifica a sintaxe de escrita da consulta. A consulta também deve ser validada verificando-se se seus atributos fazem parte do banco de dados a ser consultado. Somente então, uma representação interna da consulta é criada, geralmente representada em forma de uma arvore de consulta ou representada na forma de um grafo de consulta.

Após a tradução dessa consulta em uma estrutura que faz sentido para o SGBD uma estratégia de execução é criada. Em geral, uma consulta pode possuir várias estratégias de execuções diferentes, e o processo de escolha de uma estratégia adequada para o processamento de uma consulta é chamado de otimização de consulta (o termo "otimização" não é a melhor forma de expressar esse processo. Pois nem sempre o plano de execução escolhido é o melhor. Encontrar a otimização "perfeita" para um plano de execução pode consumir recurso e tempo escassos já que uma estratégia razoavelmente eficiente pode garantir o mesmo resultado com diferenças ínfimas de performance).

A figura 5.1 mostra diferentes passos do processamento de uma consulta de alto nível. Primeiro, pelo módulo de Tradução é criado uma representação interna da consulta em uma linguagem que o SGBD irá executar. O módulo otimizador tem a função de produzir um plano de execução, e o gerador de código gera o código que executa aquele plano. O processador em tempo de execução do banco de dados tem a função de executar o código da consulta, quer seja no modo interpretado, quer seja no modo compilado, a fim de obter o resultado da consulta. Se resultar em um erro em tempo de execução, uma mensagem de erro é gerada pelo processador em tempo de execução do banco de dados.

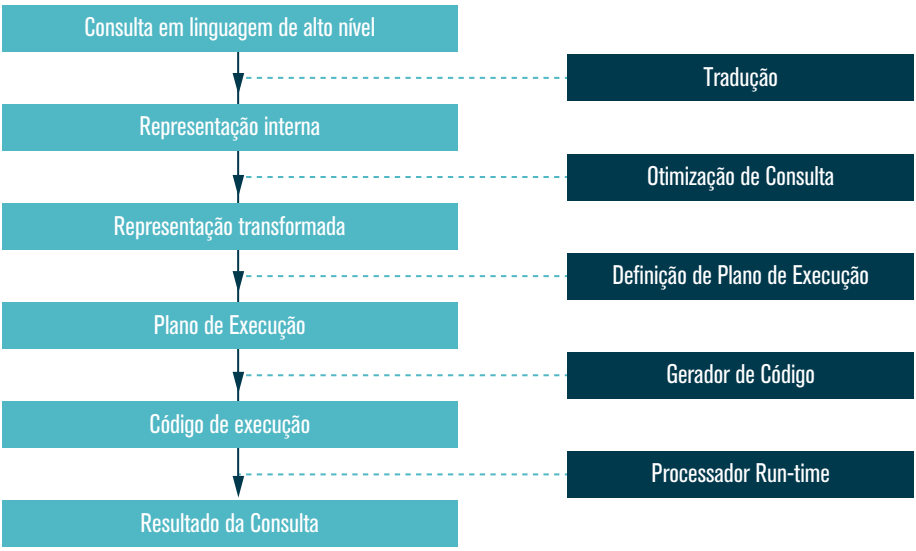


Figura 5.1 – Etapas do processamento de consultas. Fonte: Elaborada pelo autor.

5.1.1 Algoritmos para operação de Seleção

As opções de execução da operação SELECT são inúmeras. A operação é uma das mais versáteis, com várias opções de *joins* e filtros de pesquisa. Algumas opções dependem de acessos aos arquivos e por isso, só podem aplicar certos tipos de condições de seleção. A seguir são apresentados alguns dos algoritmos de busca para Seleções simples e compostas que podem ser usados para a operação de SELECT:

- Métodos de Busca para Seleções Simples:

1. Busca linear (força bruta):

Testa cada registro por vez e verifica se os valores satisfazem as condições de busca;

2. Busca binária:

Usa busca binária, que é mais eficiente que a busca linear para encontrar os valores das condições de busca;

3. Utilização de um índice primário:

Utiliza o índice primário para recuperar a informação caso o item procurado está no campo de chave única. Observe que essa condição recupera um único registro;

4. Utilização de um índice primário para recuperar múltiplos registros:

Se na condição de buscar for empregado algum dos operadores >, >=, < ou <= em um campo-chave com um índice primário, utiliza o índice primário para recuperar a informação a condição de igualdade correspondente depois recupere todos os registros seguintes no arquivo. Esta operação é também chamada de filtro, uma vez que filtra o arquivo separando os registros que não satisfazem a condição de seleção;

5. Utilização de um índice cluster para recuperar múltiplos registros:

Utiliza o índice cluster para recuperar a informação caso o item procurado não está no campo de chave única e existe um índice criado para esse campo;

6. Utilização de um índice secundário (árvore-B) em uma comparação de igualdade:

Este método de busca pode ser usado para recuperar um único registro se o campo de indexação for uma chave (possui valores únicos) ou para recuperar múltiplos registros se o campo de indexação não for chave. Ele também pode ser usado para comparações envolvendo >, >=, < ou <=.

Se em uma operação SELECT as condições de *where* são complexas e conectadas por operadores lógicos AND, temos os seguintes métodos de busca para SELECTs complexos.

- Métodos de Busca para Seleções Complexas:

7. Seleção conjuntiva utilizando um índice individual:

Se for possível usar os métodos de 2 a 6 para uma das condições separadas pelo AND, use para recuperar essa condição e em seguida verifique se cada registro recuperado satisfaz a condição da segunda parte do AND;

8. Seleção conjuntiva utilizando um índice composto:

Se for possível usar um índice composto criado com a combinação dos atributos envolvidos de cada lado do operador AND pode-se usar esse índice para recuperar a consulta diretamente;

9. Seleção conjuntiva por meio da intersecção de registros:

Se índices secundários (ou outros caminhos de acesso) estiverem disponíveis para mais de um dos campos envolvidos nas condições simples de uma condição conjuntiva, cada índice poderá ser usado para recuperar o conjunto de ponteiros de registros que satisfaça a condição individual. A intersecção desses conjuntos de ponteiros de registros resulta nos ponteiros de registros que satisfazem a condição conjuntiva e que são usados depois pata recuperar diretamente aqueles registros. Se apenas algumas das condições possuírem índices secundários, cada registro recuperado será posteriormente testado para determinar se ele satisfaz as condições restantes.

5.1.2 Algoritmos para classificação

Sempre que uma cláusula ORDER BY for adicionada a uma operação SELECT, o resultado recuperado da consulta deve ser ordenado. A ordenação também é um componente chave nos algoritmos usados no JOIN e em outras operações, tais como UNION e INTERSECTION.

Um dos algoritmos empregados na ordenação a ser usado quando ORDER BY for especificado é chamado de Algoritmo para ordenação externa (*External Sorting*). Consiste em ordenar pequenas partes também chamadas de *runs* e então juntar esses pequenos *runs* ordenados, criando partes maiores ordenadas que por sua vez são fundidas com outras partes também ordenadas, pelo fato de estar sempre ordenado e juntando pequenas partes esse algoritmo recebe o nome de *sort-merge* (ordena e junta). A figura ao lado descreve o algoritmo:

```
Inicialize:
        i = 1;
        j = b;      {tamanho do arquivo em blocos}
        k = nB;     {tamanho do buffer em blocos}
        m = ( j / k );

{Fase de Ordenação}
While (i <= m) {
        - Ler os próximos k blocos do arquivo para o buffer ou se houver menos do que k
blocos restantes, então ler os blocos restantes;
        - Ordenar os registros no buffer e gravá-los como um subarquivo temporário;
                i = i + 1;
        };

{Fase de Fusão: fundir os subarquivos até que reste apenas 1}
Inicialize:
        i = 1;
        p = logk-1 m;          {p é o número de passagens da fase de fusão}
        j = m;

        While (i <= p) {
                n = 1;
                q = (j / (k-1));

                While (n <= q) {
                -  Ler os próximos k-1 subarquivos ou os subarquivos restantes (da passa-
gem anterior), um bloco por vez;
                - Fundir e gravar como novo subarquivo um bloco por vez;
                        n = n + 1;
                }
                j = q;
                i = i + 1;
        }
```

5.1.3 Algoritmos para junção

A operação JOIN é uma das que mais consome tempo no processamento de consultas. Muitas das operações de junção encontradas nas consultas são variações de operações EQUIJOIN e NATURAL JOIN, por isso consideraremos apenas essas duas. Há muitas possibilidades de implementar uma junção de duas vias, que é uma junção em dois arquivos. As junções que envolvem mais de dois arquivos são chamadas junções de múltiplas vias. O número de maneiras possíveis para executar as junções de múltiplas vias aumenta muito ra-

pidamente. Abaixo temos as quatro técnicas mais comuns para a execução de junção de dois arquivos:

1. Junção de laços aninhados (*nested-loop*) (força bruta):

Recupera cada registro na tabela A e verifica se para cada elemento da tabela B a condição da junção é satisfeita (semelhante a um for dentro de for).

2. Junção de laço único (*single-loop*):

Se existe um índice para um dos atributos da junção na tabela A, recupera todos os registros da tabela B, e posteriormente utilize o índice para recuperar os registros que atendem a junção.

3. Junção *sort-merge* (ordenação-fusão):

Se as tabelas A e B estão fisicamente ordenados, pode-se correr os registros simultaneamente e recuperar os dados que atendem a junção. Se não estiverem ordenados, eles podem ser através de uma ordenação externa.

4. Junção *hash* (*hash-join*):

Os registros de A e B são separados em arquivos menores utilizando a mesma junção *hash* (fase de particionamento). Na segunda fase (fase de investigação) casa-se os registros correspondentes.

5.2 Otimização de Consultas

Na execução de uma consulta, quando é solicitada ao banco de dados, primeiro, ela é traduzida em uma expressão equivalente de álgebra relacional e então, é otimizada. Em geral, as consultas SQL são decompostas em blocos de consultas, que formam as unidades básicas que podem ser traduzidas em operadores algébricos e otimizados. Esse tratamento feito pelo SGBD, também conhecido como processamento de consultas, pesquisa uma expressão equivalente ao comando SQL com o objetivo de otimizar o tempo de execução da consulta. O tratamento que o SGBD realiza na execução de consultas em busca de otimização pode ser feito de duas maneiras: otimização heurística e otimização baseada em custos.

5.2.1 Heurística de Otimização de Consultas

Heurística quer dizer descobrimento, mediante o emprego de ferramentas ou procedimentos para possibilitar essa descoberta.

Uma consulta inicialmente é traduzida para a álgebra relacional, geralmente representada em uma estrutura de dados de árvore de consultas ou de um grafo de consulta. Após essa representação o analisador sintático otimiza essa consulta de acordo com regras de heurística.

Uma das principais regras heurísticas é aplicar as operações SELECT e PROJECT antes de aplicar o JOIN. As operações de SELECT e PROJECT reduzem o tamanho de um arquivo, ao passo que o JOIN ao agregar novos dados e aumentam o tamanho desse arquivo.

As regras de heurísticas devem ser aplicadas nos seguintes passos:

1. Executar operações de SELEÇÃO e PROJEÇÃO primeiramente;

2. A JUNÇÃO só deve ser realizada depois da SELEÇÃO e PROJEÇÃO;

3. Somente os atributos solicitados para o resultado da consulta e os que realmente são necessários em consultas subsequentes é que devem ser projetados;

4. Evitar geração de múltiplas tabelas intermediárias;

5. Pesquisar as subexpressões comuns e processá-las somente uma vez.

5.2.2 Medidas de Custo de uma consulta

"Um otimizador de consultas não deve depender somente de regras heurísticas; ele também deve estimar e comparar os custos da execução de uma consulta usando diferentes estratégias de execução — e deve escolher a estratégia com a menor estimativa de custo". "ELMASRI, R.; NAVATHE, S., *Sistemas de Banco de Dados* – pag. 375".

Para que a escolha de uma estratégia de execução de uma consulta seja feita, em detrimento a outra, a comparação visando os custos envolvidos deve ser feita de maneira correta e realista. Além disso as estimativas de custo levam tempo para serem analisadas, por isso deve-se empregar essa técnica em um número reduzido de estratégias. Geralmente estimativas de custos são aplicadas em consultas compiladas, onde serão usadas frequentemente pelo sistema. Para as consultas interpretadas, nas quais todo o processo ocorre em temo de execução, uma otimização em escala completa, além de consumir tempo em análise, e levando-se em conta apenas os custos de otimização, pode acabar aumentando o tempo de resposta dessas consultas. Nesse caso, uma otimização parcial, que acaba consumindo menos tempo de análise, é mais indicado para esse tipo de abordagem.

O custo da execução de uma consulta inclui os seguintes componentes:

1. Custo de acesso ao armazenamento secundário:

Esse é o custo de todo processo envolvido no acesso e armazenamento em memória secundária, geralmente o disco do computador. O processo de leitura e escrita em disco vai depender dos tipos de estruturas de acesso do arquivo gravado em disco. Se existirem índices e estruturas de busca eficientes, bem como o quão fragmentado estão os arquivos no disco, esse custo pode ser reduzido.

2. Custo de armazenamento:

Esse é o custo de armazenamento de quaisquer arquivos temporários que sejam gerados durante a execução das consultas envolvidas no processo.

3. Custo de computação:

Esse é o custo envolvido nos cálculos das operações necessárias durante a execução da consulta. Tais operações incluem a busca e ordenação das consultas, junções entre tabelas e resultados de funções envolvidas na consulta.

4. Custo do uso de memória:

Esse é o custo referente à quantidade de memória usada durante a execução da consulta.

5. Custo de comunicação:

Esse é o custo do transporte dos dados resultantes da consulta da origem para o destino. Para um sistema local, é o custo de leitura em disco e disponibilização da informação na memória ou dispositivo de saída do sistema. Mas pode ser o custo de disponibilização da informação resultante de uma consulta para um usuário final distante quilômetros da origem. Em bancos de dados distribuídos, nos quais muitos *sites* estão envolvidos, o custo de comunicação também deve ser minimizado. É difícil incluir todos esses componentes de custo em uma função de custo (ponderada) por causa da dificuldade em atribuir pesos adequados aos componentes do custo. Essa é a razão pela qual algumas funções de custo consideram apenas o acesso ao disco.

5.2.3 Análise de Plano de Execução

Um plano de execução para uma expressão de álgebra relacional representada por meio de uma árvore de consulta inclui informações sobre os métodos de acesso disponíveis para cada relação, bem como os algoritmos a serem utilizados na computação dos operadores relacionais representados na árvore.

Considere a seguinte consulta e seu plano de execução:

```
select (select ContactName from Customers where Customers.CustomerID
= Orders.CustomerID) ContactName,
                orders.OrderDate,
                Orders.OrderID,
                [Order Details].Quantity,
                ProductName
    from Orders, [Order Details],
         Products
    where [Order Details].OrderID = Orders.OrderID
         and Products.ProductID = [Order Details].ProductID
```

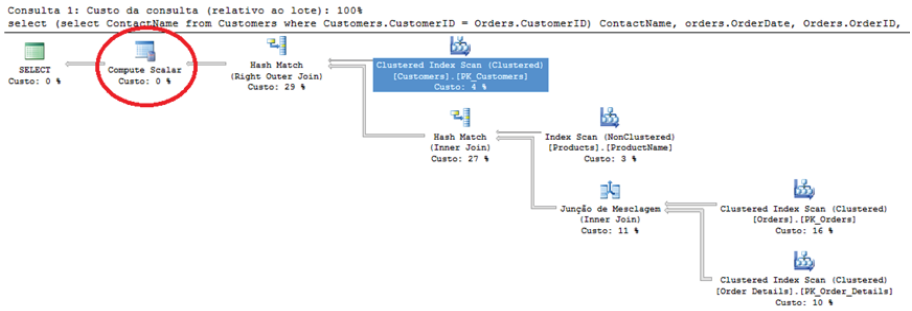


Figura 5.2 – Consulta usando subselect. Fonte: Elaborada pelo autor.

Agora, refaçamos a consulta fazendo um *join* com a tabela CUSTOMERS (sem usar o subselect):

```
select ContactName,
                orders.OrderDate,
                Orders.OrderID,
                [Order Details].Quantity,
                ProductName
    from Orders, [Order Details],
         Products,
         Customers
    where [Order Details].OrderID = Orders.OrderID
         and Products.ProductID = [Order Details].ProductID
         and customers.CustomerID = Orders.CustomerID
```

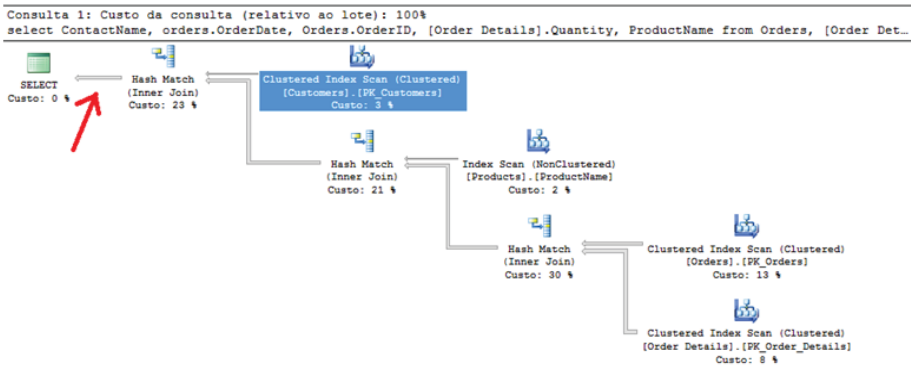


Figura 5.3 – Consuta sem subselect. Fonte: Elaborada pelo autor.

Observamos que apesar de usar um subselect e inserir uma etapa de custo 0, uma consulta sem o subselect fica mais enxuta e com menos etapas na execução. Ao analisar seu um plano de execução nota-se uma quantidade menor (no caso uma a menos) de etapas a serem executadas em uma consulta sem subselect. Pode-se então, por meio de uma geração de planos de execução, analisar diversas formas de um mesmo resultado de consulta ao SGBD e escolher o que melhor satisfaz.

5.2.4 Uso de Índices

A correta manipulação de índices é uma das maneiras mais importantes para se conseguir melhor desempenho da consulta. Esses índices agilizam a localização de dados diminuindo as operações de leitura e escrita em disco e consequentemente diminuindo o consumo de recursos do sistema.

Apenas o índice primário é criado automaticamente, desde que o projetista do banco de dados defina um campo como chave primária. Demais índices devem ser criados mediante conhecimento do projeto. É necessário entender como os dados serão usados, saber previamente quais consultas serão mais utilizadas, além de saber como o processador de consultas pode usar índices para agilizar a busca de dados.

Na escolha de índices a serem criados, é preciso examinar as consultas críticas, cujo desempenho mais afetará a experiência do usuário. Comparar o desempenho dessas consultas antes e após a criação de índices vai ajudar a

analisar se o desempenho muda com ou sem o índice. Caso a comparação não seja conclusiva ou a experiência não se mostre satisfatória, opte pela remoção do índice.

Tente descobrir o equilíbrio na criação dos índices. Lembre-se que as consultas (SELECTs) são afetadas pelos índices de maneira positiva, pois a informação indexada faz a busca ser mais rápida. Mas as operações de manipulação de dados (DML) são afetadas negativamente, pois os índices precisam ser reconstruídos sempre que uma informação é inserida (INSERT) ou removida (DELETE) de nossa base de dados. Então, se o sistema executar muitas operações DML, procure não aumentar o número de índices criados.

Para ajudar na escolha da criação de um índice é útil conhecer dois conceitos: Seletividade e Densidade. Um índice razoavelmente Seletivo e com baixa Densidade de maneira geral, é a meta a ser perseguida.

- **Seletividade:**

Um índice é dito seletivo quando ele ajuda a filtrar poucas linhas de um grande número de linhas disponíveis. O índice criado pela escolha de uma chave primária única é dito bastante seletivo, pois em uma busca pela chave, podemos garantir que apenas um registro será retornado, portanto um índice seletivo é bem-vindo no sistema.

- **Densidade:**

Um índice possui densidade elevada quando ajuda a retornar um grande conjunto de resultados em potencial. E densidade baixa quando retornar poucas linhas. Um índice criado a partir de uma chave primária terá densidade baixa, já que retorna apenas uma linha. Mas observe que para tabelas com poucas informações essa densidade pode aumentar já que sua medida é relativa a quantidade de informação existente. Portanto em uma tabela com vários registros onde se consegue criar um índice com densidade baixa esse índice pode vir a ser útil.

Conhecendo os dois conceitos anteriores, Densidade e Seletividade, fica óbvio que um índice bastante seletivo (ajudando a retornar pouca informação) e com pouca densidade (retornando pouca informação dentre um universo grande de dados) é um índice que vale a pena ser criado.

ATIVIDADES

01. Diga os principais pontos que você utilizaria para melhorar as consultas abaixo?

a) SELECT * FROM PESSOAS where CPF IS NOT NULL ORDER BY NOME

b) SELECT * FROM PESSOAS where CPF = '665.854.455-70'

02. Como utilizar um plano de execução para otimizar uma consulta?

03. Sobre a sentença: "Clusters são grupos de tabelas que ficam fisicamente armazenadas juntas porque elas compartilham colunas comuns que são frequentemente utilizadas em conjunto. Isso melhora o tempo de acesso a disco". Podemos considerar essa sentença correta?

04. Liste os principais "custos" na execução de consultas.

05. Qual o significado do termo otimização heurística? Discuta as principais heurísticas que são aplicadas durante a otimização de consultas.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] TENEMBAUM, Aaron - **Estruturas de Dados Usando C** - Ed. Person Makron Books – 2004.

[2] ELMASRI, R.; NAVATHE, S., **Sistemas de Banco de Dados**. Pearson Education do Brasil, 4ª.Ed 2005.

[3] PINHEIRO, Álvaro Farias, **Fundamentos de Engenharia de Software**: Introdução a Banco de dados, Volume II, 5ª Ed 2015.

[4] MICROSOFT TECHNET, **Ajuste do desempenho de consulta**. Disponível em: https://technet.microsoft.com/en-us/library/ms172984(v=sql.110).aspx . Acessado em: 24/04/2016.

GABARITO

Capítulo 1

01. Chave primária, chave estrangeira são conceitos importantes na modelagem de dados, pois implementam restrições que garantirá ao futuro banco de dados a integridade dos dados.

- **Chave primária**:

Atributo ou combinação de atributos que possuem a propriedade de identificar de forma única uma linha da tabela.

- **Chave estrangeira**:

A chave estrangeira ocorre quando um atributo de uma relação for chave primária em outra relação

02. Poderia ser o campo CPF ou o campo RG, pois ambos garantem um atributo único que garantem a unicidade do registro.

03. Na terminologia do modelo relacional formal, uma linha é chamada tupla, um cabeçalho de coluna é conhecido como atributo, e a tabela é chamada relação. O tipo de dado que descreve os tipos de valores que podem aparecer em cada coluna é representado pelo domínio de valores possíveis.

04. Seria necessário descobrir o CPF do Rubens ou inserir um valor único de CPF para ele se quisermos inseri-lo no banco de dados. Uma boa ideia é também ver se alguns dos CPFs já existem no banco de dados, para evitar um erro na inserção dos registros.

05. Uma tupla é uma linha de registro. Não faz sentido termos 2 linhas de registros iguais. Principalmente se existir uma chave primária definida.

Capítulo 2

01. Sim é possível. Isso é conhecido como "auto relação", Onde a chave primária de uma tabela é relacionada com uma chave secundária da mesma tabela.

02.
```
SELECT FILHOS.NOME, FILHOS.CPF, PAIS.NOME
FROM PESSOAS FILHOS, PESSOAS PAIS
WHERE PAIS.CPF = FILHOS.CPFPAI
```

03.
```
SELECT FILHOS.NOME, FILHOS.CPF, PAIS.NOME
FROM PESSOAS FILHOS, PESSOAS PAIS, PESSOAS MAES
WHERE PAIS.CPF = FILHOS.CPFPAI
AND MAES.CPF = FILHOS.CPFMAE
```

04.
```
SELECT UPPER(SOBRENOME) + ', ' + UPPER(SUBSTRING(NOME,1,1)) + LOWER(SUBSTRING(NOME,2)) AS NOME, REPLACE(REPLACE(CPF,'.',''),'-','') AS CPF
FROM PESSOAS
```

05.
```
INSERT INTO PESSOAS (NOME, SOBRENOME, CPF, CPFPAI, CPFMAE)
VALUES ('João','Silva','038.053.186-00',NULL,NULL)
```

06. A primeira soma como os números representados com os pontos decimais, irá retornar um decimal (resultado = 10.6666).

O segundo, soma números inteiros e irá retornar um inteiro desprezando a parte decimal. (resultado = 10)

07.
```
SELECT * FROM Orders,
        [Order Details],
        Products
where  [Order Details].OrderID = Orders.OrderID
        and Products.ProductID = [Order Details].ProductID
```

A resposta de qual é melhor é bastante subjetiva, mas o join feito com where e AND parece ser mais fácil de entender e contruir.

08.
```
SELECT sum(quantity), ProductName FROM  [Order Details], Products where
[Order Details].ProductID = Products.ProductID group by ProductName
having( sum(quantity) > 10)
```

Não podemos usar funções de agregação junto com colunas sem ser especificada a cláusula GROUP BY. Quando colocamos a coluna ProductName junto a uma função SUM o SQL não vai conseguir somar e mostrar os diversos nomes de produtos sem que na pesquisa seja especificado a cláusula GROUP BY. Isso quer dizer, que para qualquer coluna desejada no SELECT (desde que não seja uma função) ela deve aparecer também na cláusula GROUP BY. Além disso, se a intenção é mostrar somente os produtos cujas quantidades são maiores do que 10 por pedido, na consulta deve estar especificada a cláusula HAVING.

09. Essa é uma resposta subjetiva. Mas na opinião do Autor, a maior vantagem de se usar uma view é poder "esconder" dos usuários valores de colunas que contém informações vitais para uma empresa. Por exemplo, se existir uma tabela de funcionários e nela conter as informações de salário, o administrador de banco pode criar uma view com apenas informações básicas de funcionários e restringir o direito de leitura completa para a tabela funcionário apenas para alguns usuários da empresa.

10. As consultas em si não contêm nenhum erro. Trata-se de uma consulta usando uma subconsultas. Para cada linha de pedido ela traz o nome e o ID do cliente em questão. Ela pode ser facilmente reescrita fazendo-se o relacionamento entre as tabelas de pedidos (orders) e clientes (customers) da seguinte forma:

```
select OrderID, Customers.CustomerID, [CompanyName] from orders, Customers where
Customers.CustomerID = orders.CustomerID.
```

Capítulo 3

01. No contexto de banco de dados, um índice é uma estrutura (ou arquivo) auxiliar associado a uma tabela (ou coleção de dados). Sua função é acelerar o tempo de acesso às linhas de uma tabela, criando ponteiros para os dados armazenados em colunas específicas. O banco de dados usa o índice de maneira semelhante ao índice remissivo de um livro, verifica um determinado assunto no índice e depois localiza a sua posição em uma determinada página.
02. Referem-se aos conjuntos de um ou mais campos, cujos valores, considerando a combinação de valores em caso de mais de uma chave primária, nunca se repetem na mesma tabela e, desta forma, podem ser usadas como um índice de referência para criar relacionamentos com as demais tabelas do banco de dados.
03. Um índice secundário fornece um meio secundário de acesso a um arquivo para o qual já existe algum acesso primário. O índice secundário pode ser usado sobre um campo que é uma chave candidata e possui um valor único em cada registro, ou um campo que não é chave e que possui valores duplicados.
04. Um arquivo pode ter, no máximo, um campo de classificação física, portanto, ele só terá um índice primário ou um índice de cluster, mas não ambos.
05. Chamamos de índice denso, o índice que possui uma entrada de índice para cada registro no arquivo de dados. Um índice esparso, no entanto, tem entradas de índice para somente alguns valores de pesquisa.

Capítulo 4

01. Em um típico sistema de banco de dados, vários usuários estão fazendo acesso ao banco ao mesmo tempo. Vários aplicativos podem estar acessando os mesmos dados a cada instante. Desse modo, não é incomum que atualizações, inserções e deleções de registros estejam ocorrendo ao mesmo tempo. Dessa forma, um controle ao acesso aos dados é essencial para manter a integridade dos dados.
02. Não, pois está ocorrendo apenas uma consulta a dados sem alteração de conteúdo.
03. Sim, já que está ocorrendo uma alteração de dados, outros usuários podem estar acessando o registro. Então, para garantir a operação de ***update***, faz-se necessário o controle da transação.
04. D.
05. Sim, em alguns sistemas de banco de dados (Oracle por exemplo) é possível criar pontos de controle usando o comando SAVEPOINT.

01.

a) Se ordenações pelo campo NOME forem constantes seria interessante criar um índice para essa tabela a partir desse campo. Isso agilizaria a consulta, pois com um índice ordenado a ordenação do resultado seria mais rápido.

b) Se o campo CPF for uma chave primária, nada precisaria ser feito, já que as chaves primárias geram índices ideais para serem usados em pesquisas WHERE.

02. Podemos usar o plano de execução para "ver" como uma consulta é organizada. Testando várias consultas e analisando os planos de execução poderemos optar pela consulta com o melhor plano de execução.

03. Sim. Essa é a típica criação de um índice clustering para uma coluna bastante acessada em pesquisas que otimizará a performance das pesquisas quando fizer referência a esse conjunto de dados.

04.

a) Custo de acesso ao armazenamento secundário;

b) Custo de armazenamento;

c) Custo de computação;

d) Custo do uso de memória;

e) Custo de comunicação

05. Na otimização heurística, algumas regras são aplicadas para melhorar o desempenho da execução e da transformação das consultas em diversas expressões equivalentes. As regras são:

Executar operações de SELEÇÃO e PROJEÇÃO primeiramente;

A JUNÇÃO só deve ser realizada depois da SELEÇÃO e PROJEÇÃO;

Somente os atributos solicitados para o resultado da consulta e os que realmente são necessários em consultas subsequentes é que devem ser projetados;

Evitar geração de múltiplas tabelas intermediárias;

Pesquisar as subexpressões comuns e processá-las somente uma vez;

A utilização dessas regras heurísticas influencia no plano de execução da consulta, pois uma ordem de execução das operações é determinada e também quais os recursos serão utilizados no plano, por exemplo, os índices.