LINGUAGEM SQL/ T-SQL

SQL – STRUCTURED QUERY LANGUAGE

LINGUAGEM DECLARATIVA DE ALTO NÍVEL QUE PERMITE ESPECIFICAR O QUE SE PRETENDE COMO RESULTADO.

A OPTIMIZAÇÃO DE DECISÕES DE COMO EXECUTAR OS PEDIDOS SÃO DEIXADOS AO MOTOR DA DBMS.

CONTÉM INSTRUÇÕES PARA:
- DEFINIÇÃO DE DADOS
- CONSULTAS
- ACTUALIZAÇÃO
- definição de vistas de dados
- definição de segurança
- restrições de integridade
- controlo de transacções

1. Definição de Dados, Restrições e Esquema

1.1. Conceito de Esquema e Catálogo

Um esquema SQL agrupa as tabelas, vistas e permissões que fazem parte da BD.

É identificado por um nome, inclui um identificador do dono do esquema e os descritores dos elementos do esquema.

```
CREATE SCHEMA AUTHORIZATION owner
[ <schema_element> [...n] ]

<schema_element> ::=
{table_definition | view_definition |
grant_statement}
```

owner:	Especifica o ID do dono do esquema (conta válida na BD)
table_definition:	Especifica uma instrução **CREATE TABLE** que cria uma tabela na BD.
view_definition:	Especifica uma instrução **CREATE TABLE** que cria uma vista na BD.
grant_statement:	Especifica uma instrução **GRANT** que atribui permissões a um utilizador ou grupo.

EX:

```
CREATE SCHEMA AUTHORIZATION ross
GRANT SELECT on v1 TO public
CREATE VIEW v1(c1) AS SELECT c1 from t1
CREATE TABLE t1(c1 int)
```

O conceito de CATÁLOGO agrupa uma colecção de esquemas.

Contém um esquema especial INFORMATION_SCHEMA com informação de todos os descritores de todos os esquemas

1.2. Comando CREATE TABLE

Especifica uma nova relação – tabela, identificando-a com um nome, com os seus atributos e restrições

Primeiro são criados os atributos dando a cada: um nome, um tipo de dados para especificar a sua gama de valores, e restrições como **Not Null**.
As chaves e integridade referencial podem ser especificadas mais tarde com **ALTER TABLE**.

```
CREATE TABLE
[database_name.[owner].| owner.] table_name
({<column_definition>
 |column_name AS computed_column_expr
```

```
 |<table_constraint>::=[CONSTRAINT
constraint_name] }
 |[{PRIMARY KEY | UNIQUE} [,...n]]
)
[ON {filegroup | DEFAULT} ]
[TEXTIMAGE_ON {filegroup | DEFAULT} ]

<column_definition>::= {column_name data_type}
[[DEFAULT constant_expression ]
|[IDENTITY [(seed, increment ) [NOT FOR
REPLICATION]]]
]
[ROWGUIDCOL ]
[<column_constraint>] [ ...n]

<column_constraint>::= [CONSTRAINT
constraint_name]
{
   [NULL | NOT NULL ]
   |[{PRIMARY KEY | UNIQUE }
   [CLUSTERED | NONCLUSTERED]
   [WITH FILLFACTOR = fillfactor]
   [ON {filegroup | DEFAULT} ]]
]
|[[FOREIGN KEY]
REFERENCES ref_table [(ref_column) ]
[NOT FOR REPLICATION]
]
|CHECK [NOT FOR REPLICATION]
(logical_expression)
}

<table_constraint>::= [CONSTRAINT
constraint_name]
{
   [{PRIMARY KEY | UNIQUE }
   [CLUSTERED | NONCLUSTERED]
```

```
   {(column [ASC| DESC][,...n] ) }
   [ WITH FILLFACTOR = fillfactor]
   [ON {filegroup | DEFAULT} ]
   ]
| FOREIGN KEY
   [(column[,...n])]
   REFERENCES ref_table [(ref_column[,...n])]
   [ON DELETE {CASCADE| NO ACTION}]
   [ON UPDATE {CASCADE| NO ACTION}]
   [NOT FOR REPLICATION]
| CHECK [NOT FOR REPLICATION]
(search_conditions)
}
```

Alguns argumentos:

computed_column_expression - Expressão que define um valor calculado, é uma coluna virtual, não física, deve ser calculada a partir de outras colunas da mesma tabela.
Ex: **E_Valor as E_CUSTO * E_QT**
Podem usar-se constantes e funções.

data_type - Tipo de dados para a coluna, do sistema ou criado pelo utilizador com **sp_addtype**.

DEFAULT - Valor predefinido se não for fornecido na operação de *insert*.

IDENTITY - Cada vez que for acrescentado um registo à tabela é criado um valor único, incremental. Pode ser usado com**: tinyint, smallint, int, bigint, decimal(p,0)** ou **numeric(p,0).** Tem de especificar-se o valor inicial e o incremento, os valores predefinidos são (1,1)

NOT FOR REPLICATION - Indica que a propriedade **IDENTITY** não é passada na replicação. Os registos

replicados, na Base *Subscriber*, tem de manter o valor da base *Publisher*.

CONSTRAINT - indica o início de uma restrição: **PRIMARY KEY**, **NOT NULL**, **UNIQUE**, **FOREIGN KEY**, ou **CHECK**

As restrições a **Foreign Key** podem apenas referenciar tabelas da mesma Base no mesmo Servidor. Se a tabela estiver noutra Base terão de ser resolvidas através de *triggers*.
Trigger: tipo especial de *Stored Procedure* que é executado automaticamente quando os dados são modificados (**INSERT**, **UPDATE**, ou **DELETE**)

FOREIGN KEY...REFERENCES - Indica integridade referencial. As colunas referenciadas têm de ser chave primária ou única.

Quando há:
- inserção ou supressão de registos,
- alteração de chaves estrangeiras,

pode haver violação de integridade que deve ser resolvida, no SQL standard, com a cláusula de *trigger referencial* associada à restrição de chave estrangeira.
As opções são SET NULL, CASCADE e SET DEFAULT qualificadas com ON DELETE ou ON UPDATE.

CHECK – Restrição que obriga os valores de um campo a manterem-se numa gama.

Podem criar-se tabelas temporárias locais (**#nome**) ou globais (**##nome**)

EX:

```
CREATE TABLE DEPARTAMENTOS
(D_NOME       VARCHAR(15)    NOT NULL,
```

```
  D_NUM        INT              NOT NULL,
  D_EMP_SS     CHAR(9)          NOT NULL,
  D_EMPINI     DATE,
PRIMARY KEY (D_NUM),
UNIQUE (D_NOME),
FOREIGN KEY (D_EMP_SS) REFERENCES EMPREGADOS
(EMP_SS));
```

Tipos de Dados e DOMAINS

OS TIPOS DE DADOS PARA OS ATRIBUTOS INCLUEM TIPOS NUMÉRICOS, TEXTO; DATA E HORA

Numéricos Exactos

bigint	-2^63 .. 2^63-1
int	-2^31 (-2,147,483,648) a 2^31 - 1 (2,147,483,647)
smallint	-2^15 (-32,768) a 2^15 - 1 (32,767)
tinyint	0 a 255

bit

bit	**0 ou 1**

Decimal e numérico

decimal	**decimal**[**(***p*[**,** *s*]**)**] – Precisão: Número total de dígitos (28), com S dígitos à direita do ponto decimal. Dados: -10^38 -1 a 10^38 -1.
numeric	equivale a **decimal**.

Money and Smallmoney

money	Valores monetários de: -2^63 (-922,337,203,685,477.5808) a 2^63 - 1 (+922,337,203,685,477.5807), com precisão de décima milésima.
smallmoney	Valore monetários de: -214,748.3648 a +214,748.3647),

com precisão de décima milésima.

Numéricos Aproximados

float -1.79E + 308 a 1.79E + 308

real -3.40E + 38 a 3.40E + 38

Datetime and Smalldatetime

datetime Data e Hora desde *January* 1, 1753, a *December* 31, 9999, com precisão de 3.33 milisegundos

smalldatetime Data e Hora desde *January* 1, 1900, a *June* 6, 2079, com precisão de um minuto

Cadeias de caracteres

char Comprimento fixo, máximo de 8,000 caracteres não-Unicode

varchar Comprimento variável, máximo de 8,000 caracteres não-Unicode

text Comprimento variável, máximo de 2^31 - 1 (2,147,483,647) caracteres não-Unicode

Cadeias de caracteres Unicode

nchar Comprimento fixo, máximo de 4,000 caracteres Unicode

nvarchar Comprimento variável, máximo de 4,000 caracteres Unicode

ntext Comprimento variável, máximo de 2^30 - 1 (1,073,741,823) caracteres Unicode

Cadeias binárias

binary Comprimento fixo, máximo de 8,000 caracteres não-texto

varbinary Comprimento variável, máximo de 8,000 caracteres não-texto

image Comprimento variável, máximo de 2^31 - 1 (2,147,483,647) caracteres não-texto

Outros tipos

cursor	Referência para um cursor (não na criação de tabelas)
sql_variant	Tipo de dados que suporta vários dos tipos possíveis (excepto **text**, **ntext**, **timestamp** e **sql_variant**)
table	armazena um resultado para processamento posterior
timestamp	Valor único na BD que é actualizado sempre que o registo é actualizado. Porque no SQL 92 o significado é outro, deve usar-se o seu sinónimo: **rowversion**
uniqueidentifier	Identificador global único (GUID)

PODEM CRIAR-SE DE TIPOS DE DADOS PERSONALIZADOS COM:

CREATE DOMAIN

(Em SQL SERVER não está implementado, deve usar-se: **sp_addtype**)

Observações:

As relações criadas com **CREATE TABLE** são designadas por tabelas base (relações base) e são criadas e armazenadas pelo motor da base como um ficheiro.
Distinguem-se das relações virtuais criadas através do comando **CREATE VIEW**.

1.3. Comandos DROP SCHEMA e DROP TABLE

Se um esquema completo não for mais necessário pode ser removido com **DROP SCHEMA** (não implementado em SQL Server)

Se uma relação base não é mais necessária pode ser removida do esquema com **DROP TABLE** que tem duas opções de comportamento:

- **RESTRICT**, a tabela só é removida se não lhe - forem feitas referências
- **CASCADE**, a tabela é removida e com ela todas as Views e restrições relacionadas.

Estas opções de comportamento não estão implementadas em SQL Server – as Views e restrições tem de ser removidas uma a uma com comandos próprios.

Há uma *stored procedure*: **sp_depends** que dá informação sobre as dependências de um objecto. Mostra os objectos que dependem e os objectos dependentes de um passado como parâmetro.

1.4. Comando ALTER TABLE

Altera a definição da tabela. Permite alterar, adicionar ou remover atributos e restrições. Permite ainda activar ou desactivar restrições e *triggers*.

```
ALTER TABLE table
{ [ALTER COLUMN column_name
{ new_data_type [ (precision[, scale] ) ]
[ NULL | NOT NULL ]
| {ADD | DROP} ROWGUIDCOL }
]
| ADD
{ [ <column_definition> ]
| column_name AS computed_column_expression
}[,...n]
| [WITH CHECK | WITH NOCHECK] ADD
{ <table_constraint> }[,...n]
```

```
| DROP
{ [CONSTRAINT] constraint_name
| COLUMN column
}[,...n]
| {CHECK | NOCHECK} CONSTRAINT
{ALL | constraint_name [,...n]}
| {ENABLE | DISABLE} TRIGGER
{ALL | trigger_name[,...n]}
}
```

Ex:

Acrescenta um campo que admite **Null** (um **Not Null** tem de ter um **DEFAULT**):

```
ALTER TABLE ents
    ADD e_mbl VARCHAR(20) NULL
```

Remove um atributo:

```
ALTER TABLE contact DROP COLUMN co_telex
```

Acrescenta uma restrição a um atributo. A restrição não é validada nos registos existentes.

```
ALTER TABLE docs
    WITH NOCHECK
    ADD CONSTRAINT d_c3 CHECK (d_desc < 20)
```

2. Queries básicos

A instrução base para obtenção de informação é o **SELECT**. Permite a selecção de um ou mais registos de uma ou mais tabelas.
A sua sintaxe é complexa e deve ser estudada por fases.

```
SELECT select_list
[INTO new_table_]
FROM table_source
[WHERE search_condition]
[GROUP BY group_by_expression]
[HAVING search_condition]
[ORDER BY order_expression [ASC| DESC]]
[COMPUTE compute_expression]
[FOR for_options]
[OPTION option_hint]
```

O operador **UNION** pode ser usado entre consultas para combinar os seus resultados num só.

Cláusula Select

Especifica as colunas que devem ser incluídas no resultado

```
SELECT [ ALL | DISTINCT ]
[ TOP n [PERCENT] [ WITH TIES] ]
<select_list>

<select_list> ::=
{ *
  | {table_name|view_name|table_alias}.*
  | {column_name| expression| IDENTITYCOL|
  ROWGUIDCOL}
  [ [AS] column_alias ]
  | column_alias = expression
} [,...n]
```

Argumentos:

`All` podem aparecer linhas duplicadas na resposta (predefinido)

`DISTINCT` na resposta só aparecem linhas únicas (para este efeito o valores **`NULL`** são considerados iguais)

`Top n [Percent]` apenas as primeiras n linhas (ou as n% primeiras linhas) da resposta são mostradas. É usada a ordenação especificada.

`With TIES` São mostradas as linhas que aparecem depois das primeiras n, se tiverem o mesmo valor.

`*` Todas as colunas de todas as tabelas, indicadas nas cláusula **`FROM`** são retornadas

`column_name` Nome do atributo a obter, deve ser qualificado (antecedido do nome da tabela e de um ponto) para evitar ambiguidades.

`column_alias` Nome de substituição para o nome original ou para as expressões

Cláusula `INTO`

Cria uma nova tabela e insere os resultados do **`Select`**.

`[ INTO new_table ]`

Se no **`Select`** existir uma coluna calculada na tabela criada existirá uma coluna real.

Cláusula FROM

Especifica as tabelas de onde o resultado é obtido

```
[ FROM {<table_source>} [,...n] ]

<table_source> ::=
table_name [ [AS] table_alias ] [ WITH (
<table_hint> [,...n]) ]
| view_name [ [AS] table_alias ]
| rowset_function [ [AS] table_alias ]
| user_defined_function [ [AS] table_alias ]
| OPENXML
| derived_table [AS] table_alias
    [(column_alias [,...n] ) ]
| <joined_table>

<joined_table> ::=
<table_source> <join_type> <table_source> ON
<search_condition>
| <table_source> CROSS JOIN <table_source>
| <joined_table>

<join_type> ::=
[ INNER | { { LEFT | RIGHT | FULL } [OUTER] } ]
[ <join_hint> ]
JOIN
```

Argumentos:

OPENXML Devolve uma vista tipo tabela a partir de um documento XML. (*provider*)

rowset_function Devolve um objecto que pode ser usado no lugar de uma referência a uma tabela. (*provider*)

user_defined_function Função do utilizador que devolve um objecto que pode ser usado no lugar de uma referência a uma tabela.

table_hint Especifica um ou mais índices ou métodos a serem usados pelo Optimizador.

derived_table Instrução **Select** encaixada: *subquery*

joined_table Resultado produzido por duas ou mais tabelas.

Tipos de **join**

INNER Todas as linhas correspondentes são devolvidas (predefinido), as linhas não emparelhadas de ambas as tabelas são ignoradas.

FULL [OUTER] São incluídas todas as linhas da primeira tabela e todas as da segunda. Nas linhas em que não há correspondência aparece NULL, quer à direita, quer à esquerda.

LEFT [OUTER] São incluídas todas as linhas da primeira tabela e as correspondentes da segunda. Se estas não existirem, aparece NULL.

RIGHT [OUTER] São incluídas todas as linhas da segunda tabela e as correspondentes da primeira. Se estas não existirem, aparece NULL.

ON <search_condition> Especifica a condição em que o *Join* é feito.

CROSS JOIN São incluídas todas as linhas da primeira tabela e para cada, todas as da segunda.

Cláusula WHERE

Especifica a condição de selecção que restringe as linhas obtidas.

```
[WHERE <search_condition> ]
```

É a combinação de um ou mais predicados com os operadores lógicos AND, OR e NOT. É também usada em **Update** e **Delete**.

```
<search_condition> ::=
{[ NOT ] <predicate> | ( <search_condition> )}
[ {AND | OR}
[NOT] {<predicate> | ( <search_condition> ) } ]
} [,...n]

<predicate> ::=
{
expression { = | <> | != | > | >= | !> | < | <=
| !< } expression
|string_expression [NOT] LIKE string_expression
[ESCAPE 'escape_character']
| expression [NOT] BETWEEN expression AND
expression
| expression IS [NOT] NULL
| CONTAINS
( {column |*}, '<contains_search_condition>' )
| FREETEXT ( {column | * }, 'freetext_string' )
| expression [NOT] IN (subquery | expression
[,...n])
| expression { = | <> | != | > | >= | !> | < |
<= | !< }
{ALL | SOME | ANY} (subquery)
| [NOT] EXISTS (subquery)
}
```

Argumentos:

[NOT] LIKE Indica que a string que se segue é usada como padrão de pesquisa.

Wildcards (caracteres de substituição):
% Qualquer *string* de zero ou mais caracteres.
_ (*underscore*) Um caracter.
[ ] Um caracter na gama ([a-f]) ou colecção ([abcdef])
[^] Um caracter não na gama ([^a-f]) ou colecção ([^abcdef])

ESCAPE 'escape_character' Permite que um *wildcard* seja usado como caracter normal.

[NOT] BETWEEN Especifica um gama fechada de valores. Usa-se o **AND** para separar o inicial do final

IS [NOT] NULL Pesquisa de valores Null ou não Null.

CONTAINS Pesquisa de palavras ou frase de forma precisa ou por semelhança.

FREETEXT Pesquisa por significado e não por grafia.

[NOT] IN Pesquisa baseada no conteúdo (ou não) de uma lista – constantes ou *subquery*.

ALL Usado com um operador de relação e um *subquery* e retorna **true** se todos os valores do *subquery* satisfizerem a comparação ou **false** se não, ou se o *subquery* for vazio.

{SOME | ANY} Semelhante a **ALL** mas retorna **true** se um valor satisfizer a relação.

EXIST Usado com um *subquery* e retorna **true** se o resultado do *subquery* não for vazio.

Cláusula GROUP BY

Especifica os grupos a formar na resposta e calcula os valores resumo para cada grupo.

```
[ GROUP BY [ALL] group_by_expression [,...n]
[ WITH { CUBE | ROLLUP } ]
]
```

Argumentos:

ALL Especifica que devem ser incluídos todos os grupos, mesmo os que não tiverem linhas que verifiquem a cláusula **WHERE**.

CUBE Especifica que além do resultado gerado inclui neste todas as linhas de resumo.

Ex: Na tabela existem os registos:

Item	Color	Quantity
Table	Blue	124
Table	Red	223
Chair	Blue	101
Chair	Red	210

```
SELECT Item, Color, SUM(Quantity) AS QtySum
FROM Inventory GROUP BY Item, Color WITH CUBE
```

Item	Color	QtySum
Chair	Blue	101.00
Chair	Red	210.00
Chair	(null)	311.00
Table	Blue	124.00
Table	Red	223.00
Table	(null)	347.00
(null)	(null)	658.00
(null)	Blue	225.00
(null)	Red	433.00

ROLLUP Semelhante a CUBE, mas as linhas de resumo são apresentadas hierarquicamente pela ordem de agrupamento desaparecendo os **NULL** da primeira coluna (excepto na última).

```
SELECT CASE WHEN (GROUPING(Item) = 1)
            THEN 'ALL'
            ELSE ISNULL(Item, 'UNKNOWN')
            END AS Item,
       CASE WHEN (GROUPING(Color) = 1)
            THEN 'ALL'
            ELSE ISNULL(Color, 'UNKNOWN')
            END AS Color,
SUM(Quantity) AS QtySum FROM Inventory
GROUP BY Item, Color WITH ROLLUP
```

Item	Color	QtySum
Chair	Blue	101.00
Chair	Red	210.00
Chair	ALL	311.00
Table	Blue	124.00
Table	Red	223.00

Nota: A função **GROUPING** retorna 0 se o valor resulta dos dados e 1 se é um valor gerado pelo operador **CUBE**, onde o Null representa todos os valores

Table	ALL	347.00
ALL	ALL	658.00

Cláusula HAVING

Especifica uma condição de selecção para um grupo ou agregação. Comporta-se como um **WHERE** que opera sobre o resultado de **GROUP BY**.

```
[ HAVING <search_condition> ]
```

Operador UNION

Combina o resultado de duas ou mais consultas num único resultado com todas as linhas de cada consulta. O número e ordem das colunas de cada consulta têm de ser o mesmo e os tipos de dados compatíveis.

```
{<query specification> | (<query expression>)}
UNION [ALL]
<query specification | (<query expression>)
[UNION [ALL] <query specification | (<query
expression>)
[...n] ]
```

Argumentos:

ALL Incorpora todos as linhas no resultado, incluindo duplicados, que serão removidos, caso não seja especificado.

Cláusula ORDER BY

Especifica a ordenação do resultado.

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

Argumentos:

order_by_expression Indica o nome ou número da coluna na lista de **SELECT** a usar na ordenação.
Se existir o operador **UNION** os nomes são os do primeiro **SELECT**.

ASC ordem crescente.

DESC ordem decrescente.

Os valores **NULL** são o mais baixo possível.

Cláusula COMPUTE

Gera totais que aparecem como colunas adicionais no fim do resultado. Pode ser usado com **BY** e são geradas quebras de página e subtotais.

```
[ COMPUTE
{ { AVG | COUNT | MAX | MIN | STDEV | STDEVP
|VAR | VARP | SUM }
(expression) } [,...n]
[ BY expression [,...n] ]
]
```

Argumentos:

AVG | COUNT | MAX | MIN | STDEV | STDEVP | VAR | VARP | SUM

Especificam a agregação a fazer.

Cláusula FOR BROWSE

```
[ FOR { BROWSE | XML { RAW | AUTO | EXPLICIT }
   [ , XMLDATA ]
   [ , ELEMENTS ]
   [ , BINARY BASE64 ]
  }
]
```

Argumentos:

BROWSE Indica que o resultado pode ser actualizado pelo cliente.

XML Indica que o resultado deve ser retornado no formato XML, num dos modos especificado.

Cláusula OPTION

O optimizador das consultas cria um plano de execução para cada. Esta cláusula especifica o modo global de efectuar as consultas usado para toda a consulta. Se o optimizador não conseguir incorporar alguma das cláusulas indicadas, remove-a e recompila o plano de execução.

```
[ OPTION (<query_hint> [,...n) ]

<query_hint> ::=
{   { HASH | ORDER } GROUP
    | { CONCAT | HASH | MERGE } UNION
    | { LOOP | MERGE | HASH } JOIN
    | FAST number_rows
    | FORCE ORDER
    | MAXDOP number
    | ROBUST PLAN
    | KEEP PLAN
}
```

Argumentos:

{HASH | ORDER} GROUP Especifica que deve ser usado *Hash* ou Ordenação nos agrupamentos criados com **GROUP BY, DISTINCT** ou **COMPUTE**.

{MERGE | HASH | CONCAT} UNION Especifica o modo de efectuar as operações **UNION**.

{LOOP | MERGE | HASH} JOIN Especifica o modo de efectuar as operações **JOIN**.

FAST number_rows Especifica que a consulta é optimizada para mostrar rapidamente o número de linhas especificado e que deve continuar e determinar o resultado completo.

FORCE ORDER Especifica que a ordem indicada deve ser preservada durante a optimização.

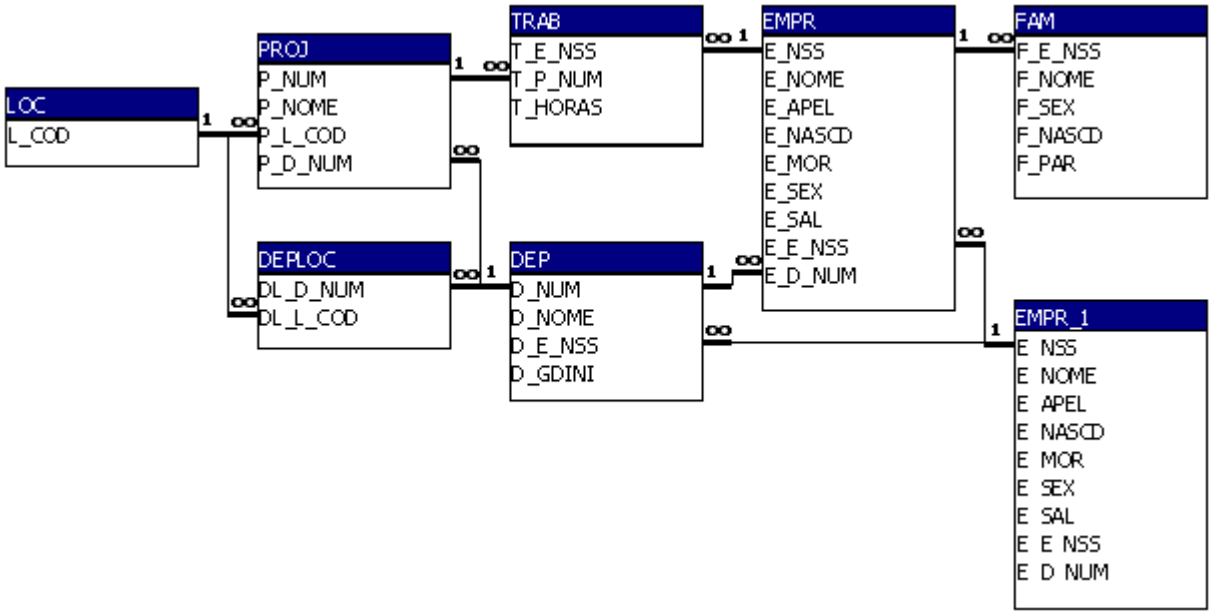
MAXDOP number Especifica o número máximo de processadores a usar pelo SQL Server numa máquina SMP *(symmetric multiprocessor*).

ROBUST PLAN Especifica que algum plano de execução que obrigue à criação de linhas (temporárias) de dimensão superior às das entradas, deve ser ignorado. Pode ter de ser gerado outro que implique maior tempo de execução.

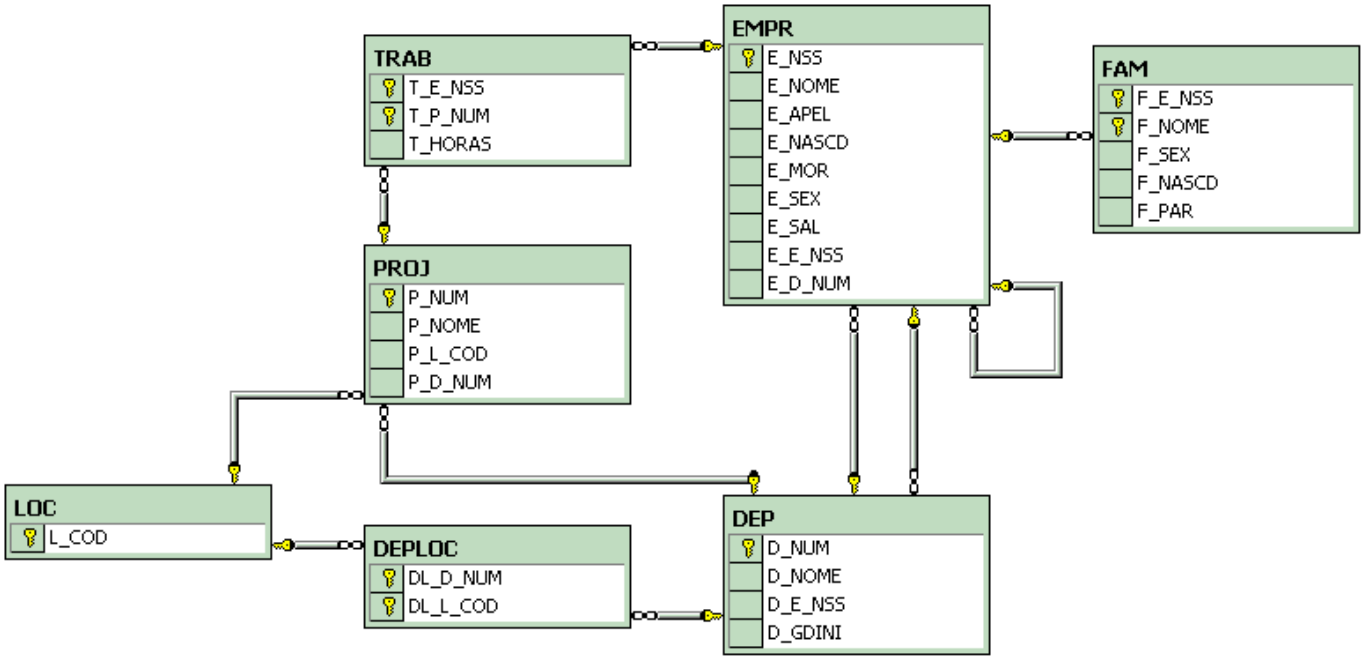
KEEP PLAN Especifica que a consulta não será recalculada sempre que haja alterações das tabelas de entrada.

2.1. Estrutura `Select - From - Where`

Consideremos uma Base de Dados com o seguinte diagrama em Access:



e o correspondente em SQL Server:



a que corresponde as definições seguintes:

```
-- tabelas

CREATE TABLE DEP (
    D_NUM       smallint NOT NULL ,
    D_NOME      nvarchar (15) NOT NULL ,
    D_E_NSS     nvarchar (9) NOT NULL ,
    D_GDINI     datetime NULL )
```

```
CREATE TABLE DEPLOC (
    DL_D_NUM   smallint NOT NULL ,
    DL_L_COD   nvarchar (12) NOT NULL )

CREATE TABLE EMPR (
    E_NSS      nvarchar (9) NOT NULL ,
    E_NOME     nvarchar (15) NOT NULL ,
    E_APEL     nvarchar (15) NOT NULL ,
    E_NASCD    datetime NULL ,
    E_MOR      nvarchar (30) NULL ,
    E_SEX      nvarchar (1) NULL ,
    E_SAL      decimal(10, 2) NULL ,
    E_E_NSS    nvarchar (9) NULL ,
    E_D_NUM    smallint NOT NULL )

CREATE TABLE FAM (
    F_E_NSS    nvarchar (9) NOT NULL ,
    F_NOME     nvarchar (15) NOT NULL ,
    F_SEX      nvarchar (1) NULL ,
    F_NASCD    datetime NULL ,
    F_PAR      nvarchar (8) NULL )

CREATE TABLE LOC (
    L_COD      nvarchar (12) NOT NULL )

CREATE TABLE PROJ (
    P_NUM      smallint NOT NULL ,
    P_NOME     nvarchar (15) NOT NULL ,
    P_L_COD    nvarchar (12) NOT NULL ,
    P_D_NUM    smallint NOT NULL )

CREATE TABLE TRAB (
    T_E_NSS    nvarchar (9) NOT NULL ,
    T_P_NUM    smallint NOT NULL ,
    T_HORAS    real not null default 0)

-- chaves

ALTER TABLE DEP ADD
     PRIMARY KEY ( D_NUM ),
     UNIQUE ( D_NOME )
ALTER TABLE DEPLOC ADD
     PRIMARY KEY ( DL_D_NUM, DL_L_COD )

ALTER TABLE EMPR ADD
     PRIMARY KEY ( E_NSS )
```

```
ALTER TABLE FAM ADD
     PRIMARY KEY ( F_E_NSS, F_NOME )

ALTER TABLE LOC ADD
     PRIMARY KEY ( L_COD )

ALTER TABLE PROJ ADD
     PRIMARY KEY ( P_NUM ),
     UNIQUE ( P_NOME )

ALTER TABLE TRAB ADD
     PRIMARY KEY ( T_E_NSS, T_P_NUM )

-- relações

ALTER TABLE DEP ADD
     FOREIGN KEY
     ( D_E_NSS ) REFERENCES EMPR ( E_NSS )

ALTER TABLE DEPLOC ADD
     FOREIGN KEY
     ( DL_D_NUM ) REFERENCES DEP ( D_NUM ),
     FOREIGN KEY
     ( DL_L_COD ) REFERENCES LOC ( L_COD )

ALTER TABLE EMPR ADD
     FOREIGN KEY
     ( E_E_NSS ) REFERENCES EMPR ( E_NSS ),
     FOREIGN KEY
     ( E_D_NUM ) REFERENCES DEP ( D_NUM )

ALTER TABLE FAM ADD
     FOREIGN KEY
     ( F_E_NSS ) REFERENCES EMPR ( E_NSS )

ALTER TABLE PROJ ADD
     FOREIGN KEY
     ( P_D_NUM ) REFERENCES DEP ( D_NUM ),
     FOREIGN KEY
     ( P_L_COD ) REFERENCES LOC ( L_COD )

ALTER TABLE TRAB ADD
     FOREIGN KEY
     ( T_E_NSS ) REFERENCES EMPR ( E_NSS ),
     FOREIGN KEY
     ( T_P_NUM ) REFERENCES PROJ ( P_NUM )
```

Vejamos alguns exemplos.

A forma básica do **SELECT** aparece com a lista de atributos a obter, a lista de tabelas de onde obter os atributos e uma expressão lógica que identifica os "registos" que farão parte da resposta.

Sql0: Obtém a data de nascimento e morada de um empregado com nome **'João Sousa'**:

```
SELECT  E_NASCD, E_MOR
FROM    EMPR
WHERE   E_NOME= 'João' AND E_APEL= 'Sousa';
```

Sql1: Obtém o nome e morada dos empregados que trabalham no departamento **'Comercial'**:

```
SELECT  E_NOME, E_MOR
FROM    EMPR, DEP
WHERE   D_NOME= 'Comercial' AND E_D_NUM= D_NUM;
```

A condição **E_D_NUM=D_NUM** equivale a um **JOIN**.

Sql2: Para os Projectos de **'Faro'** obtém o seu número, o número de Departamento e o Apelido e Data de Nascimento do Gerente do departamento:

```
SELECT  P_NUM, D_NUM, E_APEL, E_NASCD
FROM    PROJ, DEP, EMPR
WHERE   P_D_NUM = D_NUM AND D_E_NSS = E_NSS AND
        P_L_COD= 'Faro';
```

A condição de **JOIN**, **P_D_NUM = D_NUM** relaciona o Projecto com o Departamento e **D_NSS = E_NSS** relaciona o Departamento com o Empregado.

2.2. Qualificar Nomes e Renomear (Alias)

O mesmo nome pode ser usado em mais do que um atributo desde que em relações diferentes. Nas referências o nome deve ser qualificado precedendo-o do nome da relação e de um ponto.
Se no exemplo anterior em vez de **P_D_NUM** se usasse **D_NUM** teríamos:

```
SELECT  P_NUM, PROJ.D_NUM, E_APEL, E_NASCD
FROM    PROJ, DEP, EMPR
WHERE   PROJ.D_NUM = DEP.D_NUM AND D_E_NSS =
        E_NSS AND P_L_COD ='Faro';
```

Pode acontecer que a ambiguidade apareça quando a consulta se refere à mesma relação duas vezes.

Sql3: Para cada empregado obter o seu apelido e o do seu supervisor:

```
SELECT  E.E_NOME, E.E_APEL, S.E_NOME, S.E_APEL
FROM    EMPR AS E, EMPR AS S
WHERE   E.E_E_NSS = S.E_NSS;
```

2.3. *Query* sem condição e o '*'

Sql4: Se não for indicada a cláusula **WHERE** serão obtidos todos os registos:

```
SELECT  E_NSS
FROM    EMPR;
```

Sql5: Se a cláusula **FROM** tiver mais do que uma relação, o resultado é o produto cruzado de todos os registos de cada relação.

```
SELECT  E_NSS, D_NUM
```

```
FROM    EMPR, DEP;
```

Sql6: Para obter todos os atributos de uma relação pode usar-se o `'*'` em vez de os nomear a todos.

```
SELECT  *
FROM    EMPR
WHERE   E_D_NUM = 5;
```

2.4. Tabelas como conjuntos

As tabelas são tratadas não como conjuntos mas como multiconjuntos: o mesmo registo pode várias vezes aparecer numa tabela (exceto chaves únicas) tal como no resultado de um *query*. Uma tabela com chave única é um conjunto.
A eliminação de duplicados é uma tarefa não automática por:
- exigir recursos
- poder ser desejável

Para suprimir os duplicados do resultado usa-se **`DISTINCT`** na cláusula **`SELECT`**.

Sql7: Para obter o salário de cada empregado pode usar-se:

```
SELECT  E_SAL
FROM    EMPR;
```

Sql8: Para obter os diferentes salários existentes pode usar-se:

```
SELECT  DISTINCT E_SAL
FROM    EMPR;
```

Pode usar-se a operação de conjuntos **`UNION`** para obter de dois ou mais *queries* o resultado reunido de todos os registos.

Sql9: Para obtermos os projectos em que o empregado `'Cunha'` aparece como supervisor e como trabalhador poderemos usar:

```
(SELECT  DISTINCT P_NUM
FROM    PROJ, DEP, EMPR
WHERE   P_D_NUM = D_NUM AND D_E_NSS = E_NSS AND
        E_APEL = 'Cunha')
UNION
(SELECT  DISTINCT P_NUM
FROM    PROJ, TRAB, EMPR
WHERE   T_P_NUM = P_NUM AND T_E_NSS = E_NSS AND
        E_APEL = 'Cunha');
```

2.5. Comparações de *strings*, operadores aritméticos e ordenações

Sql10: Obter os empregados com morada no 'Porto':

```
SELECT  E_NOME, E_APEL
FROM    EMPR
WHERE   E_MOR LIKE '%PORTO%';
```

Sql11: Mostrar os salários dos empregados que trabalham no projecto 'Projecto X' com 10% de aumento:

```
SELECT  E_NOME, E_APEL, 1.1*E_SAL
FROM    EMPR, TRAB, PROJ
WHERE   E_NSS = T_E_NSS AND T_P_NUM = P_NUM AND
        P_NOME = 'Projecto X';
```

Sql12: Obter os empregados do departamento 5 com salário entre 10500 e 21000 Euros:

```
SELECT  *
FROM    EMPR
```

```
WHERE   (E_SAL BETWEEN 10500 AND 21000) AND
        E_D_NUM= 5;
```

Sql13: Obter uma lista de empregados e os projectos em que trabalham, ordenado por departamento e por nome em cada departamento:

```
SELECT  E_APEL, P_NOME
FROM    DEP, EMPR, TRAB, PROJ
WHERE   D_NUM = E_D_NUM AND T_E_NSS = E_NSS AND
        T_P_NUM = P_NUM
ORDER   BY D_NOME, E_APEL;
```

3. Queries mais complexos

3.1. *Queries* encadeados e Comparações de Conjuntos

Por vezes é necessário obter alguns dados e usá-los em condições de outros *queries*. Podemos encontrar na cláusula **`Where`** de um *query* (designado por exterior – *outer query*) uma estrutura **`Select – From – Where`** completa.

Sql14: O *Sql9* pode ser reformulado com um *query* encadeado:

```
SELECT  DISTINCT *
FROM    PROJ
WHERE   P_NUM IN (
        SELECT  P_NUM
        FROM    PROJ, DEP, EMPR
        WHERE   D_NUM = P_D_NUM AND D_E_NSS =
                E_NSS AND E_APEL = 'Cunha')
        OR P_NUM IN (
        SELECT  T_P_NUM
        FROM    TRAB, EMPR
        WHERE   T_E_NSS = E_NSS AND E_APEL =
                'Cunha');
```

O operador **IN** pode também ser usado para comparar um grupo de valores entre parêntesis com um conjunto de grupos de valores compatíveis:

Sql15: Selecciona o número de Segurança Social de todos os trabalhadores que trabalharam com o mesmo esquema de (Projecto, Horas) que o empregado com o número **'123456789'**:

```
SELECT  DISTINCT T_E_NSS
FROM    TRAB
WHERE   (T_P_NUM, T_HORAS) IN (
        SELECT  T_P_NUM, T_HORAS
        FROM    TRAB
        WHERE   T_E_NSS = '123456789');
```

Podem também ser usados outros operadores como **ANY | SOME** ou **ALL**.

Sql16: Obter os nomes dos empregados com salário maior do que qualquer salário dos empregados do departamento 5:

```
SELECT  E_NOME, E_APEL
FROMS   EMPR
WHERE   E_SAL > ALL (SELECT E_SAL FROM EMPR
        WHERE E_D_NUM = 5);
```

Nos *queries* encadeados é recomendável qualificar os atributos mas, se isso não for feito, a regra é o atributo ser considerado como sendo da relação mais recente.

***Queries* encadeados correlacionados:** São aqueles em a cláusula **WHERE** do *query* encadeado se refere a uma relação do *query* exterior. Deve ter-se presente que o *query* encadeado

é calculado para cada registo do *query* exterior. Por exemplo no *query*:

Sql17:

```
SELECT  E.E_NOME, E.E_APEL
FROM    EMPR AS E
WHERE   E.E_NSS IN (
        SELECT F_E_NSS
        FROM    FAM
        WHERE   E.E_NOME = FAM.F_NOME AND
                E.E_SEX = F_SEX);
```

para cada empregado, é calculado o *query* encadeado que determina o número da segurança social para todos os registos de parentes com o mesmo nome e sexo que o empregado. Se o número **E_NSS** do empregado está no resultado do *query* encadeado então o seu registo é considerado.

Modo geral um *query* encadeado com o operador **=** ou **IN** pode ser re-escrito como um único *query*.

O *Sql*17 pode ser escrito como:

Sql18:

```
SELECT  E.E_NOME, E.E_APEL
FROM    EMPR AS E, FAM AS F
WHERE   E.E_NSS = F.F_E_NSS AND
        E.E_SEX = F.F_SEX AND
        E.E_NOME = F.F_NOME;
```

3.2 Função EXISTS

A função **EXISTS** é usada para verificar se o resultado de um *query* encadeado correlacionado é vazio, ou não.

Sql19: Obtém os empregados que tenham um parente com o mesmo nome e sexo:

```
SELECT  E.E_NOME, E.E_APEL
FROM    EMPR AS E
WHERE   EXISTS (
        SELECT  *
        FROM    FAM
        WHERE   E.E_NSS = F_E_NSS AND
                E.E_SEX = F_SEX AND
                E.E_NOME = F_NOME);
```

O *query* exterior é calculado e para cada uma das suas linhas vai ser verificado se o *query* relacionado tem linhas ou não, e se tiver, a linha do *query* exterior é considerada.

O *subquery* depois de **EXISTS** não produz nenhum resultado de dados, apenas **TRUE** ou **FALSE** e por isso não faz sentido especificar quais as colunas a devolver, usa-se o '*'.

A cláusula **EXISTS** pode ser usada com **NOT**.

Sql20: Obter os nomes dos empregados sem parentes:

```
SELECT  E_NOME, E_APEL
FROM    EMPR
WHERE   NOT EXISTS (
        SELECT  *
        FROM    FAM
        WHERE   E_NSS = F_E_NSS);
```

Sql21: Obter os nomes dos gerentes com pelo menos um parente:

```
SELECT  E_NOME, E_APEL
FROM    EMPR
```

```
WHERE   EXISTS (
        SELECT  *
        FROM    FAM
        WHERE   F_E_NSS = E_NSS)
        AND EXISTS (
        SELECT  *
        FROM    DEP
        WHERE   D_E_NSS = E_NSS);
```

3.3 Conjuntos explícitos e NULLS

Na cáusula **WHERE** podem ser especificadas várias constantes em vez do *subquery*.

Sql22: Obter o número da segurança social dos empregados que trabalham nos projectos **1**, **2** e **3**:

```
SELECT  DISTINCT E_NSS
FROM    TRAB
WHERE   T_P_NUM IN (1,2,3);
```

Sql23: Obter os nomes dos empregados que não têm supervisor:

```
SELECT  E_NOME, E_APEL
FROM    EMPR
WHERE   E_E_NSS IS NULL;
```

Note-se que com **NULL** se usa **IS** em vez do '**=**' e **IS NOT** em vez do '**<>**', isto porque o SQL considera cada **NULL** diferente de outro.

3.4 Renomear atributos e tabelas JOIN

Podem renomear-se os atributos que aparecem como resultado de um *query* usando o qualificador **AS** seguido do novo nome

pretendido. Os novos nomes irão aparecer como título de colunas no resultado.

Sql24:

```
SELECT  E.E_APEL AS NOME_EMPREGADO, S.E_APEL AS
        NOME_SUPERVISOR
FROM    EMPR AS E, EMPR AS S
WHERE   E.E_E_NSS = S.E_NSS;
```

O conceito de tabelas **JOIN** foi introduzido no SQL92 para permitir aos utilizadores a especificação de uma tabela que resulte da operação de junção na cláusula **FROM** separando na cláusula **WHERE** o que são as condições de selecção do que são as condições de junção.

Sql25: Podemos re-escrever o *Sql1*, que obtém o nome e morada dos empregados que trabalham no departamento **'Comercial'**. Em vez de:

```
SELECT  E_NOME, E_MOR
FROM    EMPR, DEP
WHERE   D_NOME = 'Comercial' AND E_D_NUM =
        D_NUM;
```

Podemos escrever:

```
SELECT  E_NOME, E_MOR
FROM    (EMPR JOIN DEP ON E_D_NUM = D_NUM)
WHERE   D_NOME = 'Comercial';
```

O tipo predefinido de um **JOIN** é o **INNER JOIN** onde as linhas que são incluídas na resposta são as que correspondem a registos que existam em ambas as relações. Os registos com **NULL** são excluídos.

Podem usar-se **OUTER JOIN** para obter na resposta os registos sem correspondência na primeira ou segunda relação ou mesmo em ambas.
Um **LEFT JOIN** mantém todos os registos da 1ª relação, ou relação à esquerda.
Um **RIGHT JOIN** mantém todos os registos da 2ª relação, ou relação à direita.
Para manter todos os registos das duas relações usa-se **FULL OUTER JOIN**.

Sql26: O *Sql24* que obtinha o nome de cada empregado e do seu supervisor só mostrava como resultado os empregados que tivessem supervisor. Pode ser modificado para obter todos os empregados e os supervisores que existam:

```
SELECT  E.E_APEL AS NOME_EMPREGADO,
        S.E_APEL AS NOME_SUPERVISOR
FROM    (EMPR AS E LEFT OUTER JOIN EMPR
        AS S ON E.E_E_NSS = S.E_NSS);
```

É possível encadear **JOIN**, ou seja uma das relações pode ser o resultado de um **JOIN**.

Sql27: Vimos no *Sql2* uma forma de obter para os Projectos de **'Faro'** o seu número, o número de Departamento e o Apelido e Data de Nascimento do Gerente do departamento:

```
SELECT  P_NUM, D_NUM, E_APEL, E_NASCD
FROM    PROJ, DEP, EMPR
WHERE   P_D_NUM = D_NUM AND D_E_NSS = E_NSS AND
        P_L_COD = 'Faro';
```

Podemos re-escrevê-lo com o conceito de **JOIN**.

```
SELECT  P_NUM, D_NUM, E_APEL, E_NASCD
```

```
FROM    ((PROJ JOIN DEP ON P_D_NUM = D_NUM)
        JOIN EMPR ON D_E_NSS = E_NSS)
WHERE   P_L_COD = 'Faro';
```

3.5 Funções de agregação e agrupamento

As funções **COUNT**, **SUM**, **MAX**, **MIN** a **AVG** podem ser usadas nas cláusulas **WHERE** e **HAVING**.

Sql28: Determinar a soma dos salários de todos os empregados e o máximo, o mínimo e a média:

```
SELECT  SUM(E_SAL), MAX(E_SAL), MIN(E_SAL),
        AVG(E_SAL)
FROM    EMPR;
```

Sql29: Podemos restringir a resposta aos empregados do departamento `'Comercial'`:

```
SELECT  SUM(E_SAL), MAX(E_SAL), MIN(E_SAL),
        AVG(E_SAL)
FROM    (EMPR JOIN DEP ON E_D_NUM = D_NUM)
WHERE   D_NOME= 'Comercial';
```

Sql30: Conta o número de empregados:

```
SELECT  COUNT (*)
FROM    EMPR;
```

Sql31: Conta o número de empregados do departamento `'Comercial'`:

```
SELECT  COUNT (*)
FROM    (EMPR JOIN DEP ON E_D_NUM = D_NUM)
WHERE   D_NOME= 'Comercial';
```

O `'*'` refere-se às linhas do resultado.

Sql32: Conta o número de salários diferentes:

```
SELECT  COUNT (DISTINCT E_SAL)
FROM    EMPR;
```

Sql33: Para obter o nome dos empregados com mais do que um familiar:

```
SELECT  E_NOME, E_APEL
FROM    EMPR
WHERE   ( SELECT COUNT (*)
        FROM FAM
        WHERE E_NSS= F_E_NSS) >=2;
```

Podemos aplicar as funções a subgrupos de linhas da relação formados com base nalguns atributos.

Sql34: Para cada departamento obter o seu número, a quantidade de empregados e a média dos seus salários:

```
SELECT  E_D_NUM, COUNT(*), AVG (E_SAL)
FROM    EMPR
GROUP BY  E_D_NUM;
```

Sql35: Para cada projecto obter o seu número e nome e a quantidade de trabalhadores que a ele se dedica:

```
SELECT  P_NUM, P_NOME, COUNT(*)
FROM    (PROJ JOIN TRAB ON P_NUM = T_P_NUM)
GROUP BY  P_NUM, P_NOME;
```

Quando se usa `JOIN` as funções e agrupamento são aplicados depois da execução da junção das relações.

Podemos estar interessados apenas nos agrupamentos que satisfaçam alguma condição e para isso usamos a cláusula **HAVING**. Opera como uma selecção sobre o agrupamento.

Sql36: Para cada projecto, com mais de dois trabalhadores, obter o seu número e nome e a quantidade de trabalhadores que a ele se dedica:

```
SELECT  P_NUM, P_NOME, COUNT(*)
FROM    (PROJ JOIN TRAB ON P_NUM = T_P_NUM)
GROUP BY  P_NUM, P_NOME
HAVING  COUNT(*)> 2;
```

Outra forma de ver este *query* seria traduzir o **JOIN** pela cláusula **WHERE**:

```
SELECT  P_NUM, P_NOME, COUNT(*)
FROM    PROJ, TRAB
WHERE   P_NUM = T_P_NUM
GROUP BY  P_NUM, P_NOME
HAVING  COUNT(*)> 2;
```

e pode observar-se melhor como o **WHERE** limita as linhas às quais as funções são aplicadas.

Sql37: Para cada projecto obter o seu número e nome e a quantidade de trabalhadores do departamento 5 que a ele se dedica:

```
SELECT  P_NUM, P_NOME, COUNT(*)
FROM    PROJ, TRAB, EMPR
WHERE   P_NUM= T_P_NUM AND E_NSS= T_E_NSS AND
        D_NUM=5
GROUP BY  P_NUM, P_NOME;
```

Ou usando **JOIN**:

```
SELECT  P_NUM, P_NOME, COUNT(*)
FROM    (PROJ JOIN (EMPR JOIN TRAB ON E_NSS=
        T_E_NSS) ON P_NUM= T_P_NUM)
WHERE   D_NUM=5
GROUP BY  P_NUM, P_NOME;
```

4. Instruções INSERT, DELETE e UPDATE

4.1 O Comando INSERT

Na forma mais simples junta um registo a uma relação:

Sql38: Os valores têm de ser indicados pela ordem do **CREATE TABLE** correspondente à relação.

```
INSERT  INTO EMPR
VALUES  ('GIL', 'DIAS', '123456789', '1973-06-
        27', 'R. FEZ, 123 - 4100-783 PORTO',
        'M', 22500, '987654321', 5);
```

Uma segunda forma permite ao utilizador explicitar os atributos, pela ordem desejada, e os correspondentes valores:

Sql39: Têm de, pelo menos, ser especificados todos os atributos **NOT NULL** sem valor predefinido:

```
INSERT  INTO EMPR (E_NOME, E_APEL, E_D_NUM,
        E_NSS)
VALUES  ('RUI', 'PINTO', 4, '111222333');
```

Os registos que não verificarem as restrições, quer de atributos, quer de integridade, serão rejeitados.

Uma outra forma permite juntar uma colecção de registos. Por exemplo podemos criar uma tabela temporária para armazenar o nome, número de empregados e total de salários para cada departamento.

Sql40:

```
CREATE TABLE DEP_R
(  DR_NOME    VARCHAR(15),
   DR_NE      INTEGER,
   DR_TSAL    INTEGER);
```

Sql41:

```
INSERT  INTO DEP_R (DR_NOME, DR_NE, DR_TSAL)
SELECT  D_NOME, COUNT(*), SUM(E_SAL)
FROM    (DEP JOIN EMPR ON D_NUM = E_D_NUM)
GROUP BY  D_NOME;
```

Nota: Se registos da tabela **EMPR** forem alterados a tabela **DEP_R** fica desactualizada.

4.2 O Comando DELETE

Remove registos da relação. Inclui uma cláusula **WHERE** para selecção. Se forem especificados **TRIGGERS** referenciais a remoção pode propagar-se a outras relações. É possível apagar 0, 1, vários ou todos os registos, dependendo da selecção.

Sql42: Se não existir nenhum empregado com apelido **'Rodrigues'** nenhuma acção é desencadeada:

```
DELETE  FROM EMPR
WHERE   E_APEL= 'Rodrigues';
```

Sql43: Para remover um empregado com número de segurança social `'345733456'`:

```
DELETE   FROM EMPR
WHERE    E_NSS= '345733456';
```

Sql44: Para remover os empregados dos departamentos 7 e 8:

```
DELETE   FROM EMPR
WHERE    E_D_NUM IN (7,8);
```

Sql45: Para remover todos empregados:

```
DELETE   FROM EMPR;
```

ou

```
DELETE EMPR;
```

4.3 O Comando `UPDATE`

Modifica o valor dos atributos numa ou mais linhas seleccionadas de uma relação. Se for necessário modificar várias relações terão de ser escritos vários comandos **`UPDATE`**.
A modificação de uma chave primária numa relação pode propagar-se a outras relações se estiverem especificadas as restrições de integridade necessárias.
Aparece uma cláusula **`SET`** onde é indicada a lista dos atributos e dos seus novos valores.

Sql47: Para passar o projecto `10` para a localização `'TROFA'` e para o departamento `5`:

```
UPDATE   PROJECT
SET      P_L_COD = 'TROFA', P_D_NUM = 5
WHERE    P_NUM =10;
```

É possível modificar várias linhas com um único comando.

Sql48: Para aumentar o salário de todos os empregados do departamento `'Qualidade'` em `6%`:

```
UPDATE  EMPR
SET     E_SAL = E_SAL*1.06
WHERE   E_D_NUM IN (SELECT D_NUM FROM DEP
                    WHERE D_NOME = 'Qualidade');
```

5. VIEWS – Tabelas Virtuais

5.1. Conceito de VIEW

Uma **`VIEW`** é uma tabela virtual única derivada de outras tabelas ou *views* com algumas limitações nas modificações aos valores dos atributos mas sem restrições a consultas.
É uma forma de especificar uma colecção de atributos de que precisamos com frequência. Em vez de termos de especificar de todas as vezes todas as relações e **`JOINS`** criamos uma **`VIEW`**. Os dados disponíveis com uma **`VIEW`** estão sempre actualizados.

5.2. Criação de `VIEWS`

Sql49: Cria uma **`VIEW`** com o nome, apelido do empregado, nome de projecto e esquema de horas:

```
CREATE VIEW EMPRPROJ
AS  SELECT E_NOME, E_APEL, P_NOME, T_HORAS
    FROM PROJ INNER JOIN (EMPR INNER JOIN TRAB
       ON E_NSS = T_E_NSS) ON P_NUM = T_P_NUM;
```

Sql50: Cria uma **`VIEW`** com o nome dos departamentos, quantidade de empregados e total de salários de cada, com a especificação de novos nomes para atributos:

```
CREATE VIEW DEPINF (DI_NOME, DI_QE, DI_TSAL)
AS  SELECT D_NOME, COUNT(*), SUM(E_SAL)
    FROM DEP INNER JOIN EMPR ON D_NUM = E_D_NUM
                GROUP BY D_NOME;
```

Os novos nomes são obrigatórios sempre que há campos calculados.

Uma **VIEW** é actualizável se:
- a cláusula **SELECT** não contém funções de agregação nem as cláusulas **TOP**, **GROUP BY**, **UNION**, ou **DISTINCT**,
- a cláusula **SELECT** não contém colunas derivadas de outras – expressões,
- a cláusula **FROM** refere pelo menos uma tabela.

6. Stored Procedures (SQL Server)

Uma *stored procedure* – ***SP***, é uma colecção de instruções **T-SQL** que se armazena com a Base de Dados e que encapsula uma tarefa que se realizará várias vezes. Nas ***SP*** podemos declarar variáveis, usar instruções de repetição condicional, aceitar parâmetros e retornar valores.
No SQL Server existem 5 tipos:
- Sistema (**sp_**): armazenadas na base de dados **master** constituem uma forma expedita de obter informação das tabelas de sistema.
- Locais: Criadas nas bases de dados do utilizador.
- Temporárias: Podem ser locais (nome começa por **#**) e são válidas apenas na sessão corrente, ou globais (o nome começa por **##**) e ficam disponíveis para todas as sessões.
- Remotas: Permitem a execução de ***SP*** em servidores remotos.
- Externas: Permitem a execução de ***SP*** fora do ambiente SQL Server através de DLLs.

6.1. Considerações

Da primeira vez que uma ***SP*** é executada, ou quando é compilada, o processador de *queries* efectua uma verificação sintáctica. Depois cria um plano de execução ou seja determina a forma mais rápida de aceder aos dados – fase de Optimização, levando em conta:

- Quantidade de dados nas tabelas
- Existência de índices
- Operadores de relação e valores na cláusula **`WHERE`**
- Existência de cláusulas **`JOINS`**, **`UNION`**, **`GROUP BY`** e **`ORDER BY`**

O resultado desta análise resulta no armazenamento de uma versão compilada – fase de Compilação.

6.2. Vantagens

- Partilha de lógica assegurando a consistência de acesso e modificação dos dados. As regras de negócio e funções podem ser tratadas de uma forma segura apenas num local.
- Se as ***SP*** suportarem todas as regras e funções de negócio os utilizadores podem não ter necessidade de manipular as tabelas directamente.
- É possível dar acesso a ***SP*** sem que o utilizador tenha acesso às tabelas ou **`VIEWS`**.
- A eficiência é melhorada pelo estabelecimento prévio de planos de execução.
- O tráfego na rede baixa pelo facto de ser passada apenas uma instrução simples.

6.3. Criação de Stored Procedures.

A concepção de uma **`SP`** é semelhante à de um **`VIEW`**. Escrevem-se e testam-se as instruções SQL e depois de correctas incluem-se na **`SP`**. Usa-se:

```
CREATE PROC[EDURE] procedure_name [;number]
[ {@parameter data_type}
    [VARYING] [= default] [OUTPUT]
][,...n]
[WITH {RECOMPILE | ENCRYPTION | RECOMPILE,
ENCRYPTION } ]
[FOR REPLICATION] AS sql_statement [...n]
```

number – Permite agrupar no mesmo nome diferentes **SP** de modo a poderem ser **DROPed** em simultâneo.

@parameter – Os parâmetros são locais à **SP**. Podem assumir o lugar de constantes, não de tabelas, atributos nem de outros objectos da base de dados.

data type – Pode ser qualquer um, embora o tipo cursor só possa ser usado como parâmetro de saída.

VARYING – O cursor resultante pode variar.

OUTPUT – O parâmetro é de saída.

RECOMPILE - O plano de execução não é armazenado.

ENCRYPTION - O texto da **SP** é armazenado criptado.

FOR REPLICATION – As **SP** criadas para replicação não são executadas no Servidor Subscritor.

sql_statement – As instruções SQL que constituem a **SP**.

Alguns factos com as **SP**:

- Podem referenciar tabelas, **VIEWs**, **SPs** e tabelas temporárias

- Uma tabela local temporária criada numa **SP** só existe durante a execução da **SP**.
- A definição de uma **SP** pode incluir qualquer número de instruções SQL excepto alguns **CREATEs**.
- Uma **SP** chamada de outra pode referenciar todos os seus objectos.

6.4. Execução de Stored Procedures.

As **SP** podem ser executadas autonomamente com:

```
[[EXEC[UTE]]
{[@return_status =] {procedure_name [;number] |
@procedure_name_var }
[[@parameter =] {value | @variable [OUTPUT] |
[DEFAULT]] [,...n]
[WITH RECOMPILE]
```

@return_status – Variável inteira que recebe o estado resultante da execução.

@procedure_name_var – Nome de uma variável local que representa o nome da ***SP***.

@parameter – Nome dos parâmetros, por serem nominativos podem ser especificados por qualquer ordem.

Alternativamente uma **SP** pode ser executada como parte da instrução **INSERT INTO** para fornecer a colecção de linhas a inserir numa tabela.

6.5. Alterar e Remover Stored Procedures.

As ***SP*** podem ser redefinidas com a instrução **ALTER PROCEDURE**, que não tem efeito em **SP** ou em **TRIGGERS** dependentes. A sintaxe é semelhante à de **CREATE**.

Para remover uma ou mais **SP** usa-se:

```
DROP PROCEDURE {procedure} [,...n]
```

A **SP** de sistema **sp_depends** permite saber se há outras **SP** que dependam da que se pretende remover.

7. Triggers (SQL Server 2000)

Um *Trigger* é um tipo especial de **SP** que é executado sempre que os dados da tabela ou vista associada são modificados.

Factos com *Triggers*:
- Um *Trigger* está associado a uma **TABLE** ou **VIEW**
- A sua execução é automática quando os registos são modificados com **INSERT**, **UPDATE** ou **DELETE** se existir um *Trigger* definido para essa acção.
- Não podem ser chamados directamente e não aceitam nem devolvem parâmetros.
- O *Trigger* e a instrução que o activa são considerados como uma única transacção.

7.1. Uso de Triggers

A sua principal utilização é para assegurar regras de negócio ou requisitos complexos. Exemplos:
- Propagar alterações em tabelas relacionadas. Quando se modifica ou insere um registo numa tabela é possível localizando os registos relacionados, de outras tabelas, modificá-los também.
- Forçar uma restrição mais complexa do que a obtida com **CHECK**:

  - Os *Triggers* podem referenciar atributos de outras tabelas e podem assegurar acções complexas quando se devem propagar operações de **DELETE** ou **UPDATE**.
  - Podem modificar mais do que um registo.
  - Forçar integridade referencial entre várias bases de dados.
- Permite definir mensagens do utilizador para situações anómalas.
- Permite manter dados não normalizados, tipicamente tratar de redundâncias e campos calculados.

Factos a ter em conta:
- São reactivos às instruções **INSERT**, **UPDATE** e **DELETE**.
- Uma tabela pode ter vários *Triggers* para uma mesma acção.
- Não podem ser criados para tabelas temporárias.
- Não podem devolver registos.
- Podem tratar acções multiregisto processando todos os registos ou usando acções condicionais.

7.2. Definição de Triggers

Os *Triggers* são criados com:

```
CREATE TRIGGER trigger_name
ON {table | view} [WITH ENCRYPTION]
{
 {{FOR|AFTER|INSTEAD OF}
   {[DELETE][,][INSERT][,][UPDATE]}
  [WITH APPEND]
  [NOT FOR REPLICATION]
  AS
  [{IF UPDATE (column)
    [{AND|OR} UPDATE (column)] [...n]
  |IF (COLUMNS_UPDATED ( ) {bitwise_operator}
        updated_bitmask)
       {comparison_operator} col_bitmask [...n]
  }]
```

```
  sql_statement [...n]
 }
}
```

Alguns argumentos:

WITH ENCRYPTION Cripta o texto do *trigger* impedindo que ele seja publicado como parte da replicação

AFTER O *trigger* é executado depois de terminada a instrução, incluindo as **CONSTRAINTS**, que o disparou. Se a instrução falhou, p.ex. com erro integridade, o *trigger* não é executado.
É o tipo predefinido no SQL 2000 e o único nas versões anteriores.
Podem existir vários para a mesma acção e o primeiro e último podem ser estipulado com **sp_settriggerorder**.
Só pode ser criado para **TABLE**.

INSTEAD OF O *trigger* é executado em vez da acção que o dispara, logo, antes das **CONSTRAINTS**.
Podem ser definidos para **TABLES** e **VIEWS**, mas apenas um para cada acção.
São usados para permitir que **VIEWS** formadas a partir de várias tabelas, possam suportar as acção de **INSERT**, **DELETE** e **UPDATE**.

{[DELETE][,][INSERT][,][UPDATE]} São as acções para as quais os *triggers* estão definidos. É obrigatório indicar uma mas podem ser indicadas todas e por qualquer ordem.
Para os *triggers* **INSTEAD OF** a opção **DELETE** não pode ser indicada se as regras de integridade referencial especificarem uma acção em cascata para o **DELETE**. De igual modo **UPDATE** não pode ser

indicado se as regras de integridade tiverem uma acção em cascata para o **UPDATE**.

IF UPDATE (column) Para testar se uma coluna em particular foi alvo de **UPDATE** ou **INSERT** (pode ter valores implícitos e não ser incluída na acção)

IF COLUMNS_UPDATED Testa se no **INSERT** ou **UPDATE** as colunas mencionadas foram incluídas

Notas:

Os *triggers* são usados para assegurar regras de negócio e de integridade dos dados. Em SQL há um mecanismo: DRI (*declarative referential integrity*) que assegura a integridade; o uso de *triggers* para esse efeito só deve ocorrer para regras de integridade externas (*cross-database referential integrity*).

As **CONSTRAINTS** da tabela do *trigger* são verificadas depois do **INSTEAD OF** e antes do **AFTER** e se forem violadas as acções desencadeadas no **INSTEAD OF** são desfeitas e o **AFTER** não é disparado.

O **AFTER** vê todas as alterações feitas na tabela pela acção que o disparou, incluindo as alterações em cascata.

Algumas instruções SQL não podem ser incluídas na definição dos *Triggers*.

7.3. Alterar e Remover Triggers

A alteração de um *Trigger* substitui a sua definição corrente por outra e é feita com **ALTER TRIGGER** que tem uma sintaxe semelhante à da criação.
Pode também alterar-se a acção inicial que o activava.

Para remover um *Trigger* usa-se **DROP TRIGGER**.

7.4. Funcionamento dos *Triggers*

Quando um *trigger* **INSERT** é activado os registos novos foram adicionados à tabela associada e são igualmente adicionados a uma tabela lógica inserted onde os registos podem ser acedidos para desencadear novas acções

Quando um *trigger* **DELETE** é activado os registos removidos da tabela associada são passados para uma tabela lógica deleted onde podem continuar a ser acedidos.

A instrução de **UPDATE** pode ser vista como tendo dois passos: um **DELETE** que capta a imagem inicial e um **INSERT** que capta a imagem final dos dados.
Quando um *trigger* **UPDATE** é activado as linhas originais são passadas para a tabela deleted e as linhas depois de modificadas são copiadas para a tabela inserted. O *trigger* pode examinar ambas as tabelas e determinar novas acções.

Um *trigger* pode conter instruções **UPDATE**, **INSERT** ou **DELETE** que afectem outras tabelas e pode assim desencadear outros *triggers*. Podem existir até 32 níveis de desencadeamento. Verifica-se que:

- Um *trigger* não é activado duas vezes na mesma transacção, ou seja, não se chama a ele próprio em resposta a uma segunda actualização da mesma tabela. Ou dito ainda de outra forma, se um *trigger* modifica uma tabela que por sua vez modifica a tabela do *trigger* inicial, o *trigger* não é activado de novo.
- Por um *trigger* ser uma transacção, uma falha em qualquer nível dos *triggers* desencadeados cancela toda a transacção e todas as modificações são desfeitas.

O desencadeamento de *triggers* pode ser suspenso com:

```
sp_configure 'nested triggers', 0
```

7.4.1. *Triggers* Recursivos

Com a possibilidade do uso da recursividade um *trigger* que modifique os dados de uma tabela pode activar-se a si próprio. Esta possibilidade está suspensa a não ser que se use:

```
sp_dboption databasename, 'recursive triggers', True
```

A recursividade pode ser directa ou indirecta e pode ocorrer até 32 níveis.

7.5. Exemplo com *Trigger*

Uma das situações em que se pode usar *triggers* é para manter a integridade dos dados.
Vejamos um exemplo onde existe uma tabela de reservas de livros e outra de empréstimos.

EMPR

	ISBN	COP	MEMB
	1	1	1
⇨	4	1	7
	4	2	4
	3	1	3
	1	2	1

✂

RES

ISBN	MEMB	DATA
1	1	14/3
1	2	12/3
2	1	4/5
3	1	7/4
4	7	21/7

Ao inserir-se na tabela dos **EMPR**éstimos um registo correspondente a uma **RES**erva, esta deve ser eliminada.
Podemos escrever o seguinte *trigger*:

```
CREATE TRIGGER RES_REM
    ON EMPR
    FOR INSERT AS
```

```
IF    (SELECT RES.MEMB FROM RES JOIN inserted
      AS i ON RES.MEMB= i.MEMB AND RES.ISBN=
      i.ISBN) >0
BEGIN
      DELETE RES FROM RES INNER JOIN inserted
      AS i ON RES.MEMB= i.MEMB AND RES.ISBN=
      i.ISBN
END
```

7.6. Considerações finais

- Os *triggers* são rápidos uma vez que as tabelas inserted e deleted trabalham em memória.
- O tempo de execução depende dos registos afectados e das tabelas que estiverem envolvidas.
- As acções contidas num *trigger* fazem parte de uma transacção única.

8. Transacções (SQL Server)

O mecanismo das transacções assegura que várias alterações aos dados são processadas como uma unidade de trabalho. Cada unidade tem de ser executada completamente ou não é executada de todo.
Por exemplo um movimento bancário deve debitar uma conta e creditar outra e os dois passos só fazem sentido se ambos tiverem sucesso.

Cada unidade de trabalho tem de apresentar 4 propriedades: ACID (Atomicidade, Consistência, Isolamento e Durabilidade) para poder ser considerada uma transacção.

Atomicidade: todas as modificações são efectuadas, ou nenhuma ocorrerá

Consistência: Depois de terminada a transacção deve deixar os dados num estado consistente, todas as regras de dados

se devem verificar e todas as estruturas internas (B-trees, listas, ...) devem estar correctas

Isolamento: Existindo transacções concorrentes, cada uma vê os dados, antes ou depois das outras terem terminado e nunca num estado intermédio - serialização

Durabilidade: depois de a transacção ter terminado as modificações são definitivas no sistema, mesmo que ocorram falhas.

8.1. Transactions

Há dois tipos de transacções:

- Explícitas, ou definidas pelo utilizador: Agrupam instruções entre:

  `BEGIN {TRAN[SACTION] | WORK}` e

  `COMMIT {TRAN[SACTION] | WORK}`

  que opcionalmente podem ter um nome.

  Uma transacção pode ser cancelada ou desfeita:

  `ROLLBACK {TRAN[SACTION] | WORK}`

- Implícitas: A sessão é iniciada neste modo com:

  `SET IMPLICIT_TRANSACIONS ON`

  A partir daqui é gerada uma cadeia de transacções cada uma iniciada com **`CREATE TABLE, CREATE, DELETE, DROP, FETCH, GRANT, REVOKE, OPEN, INSERT, TRUNCATE, UPDATE`** e **`SELECT`** e terminada com **`COMMIT`** ou **`ROLLBACK.`**

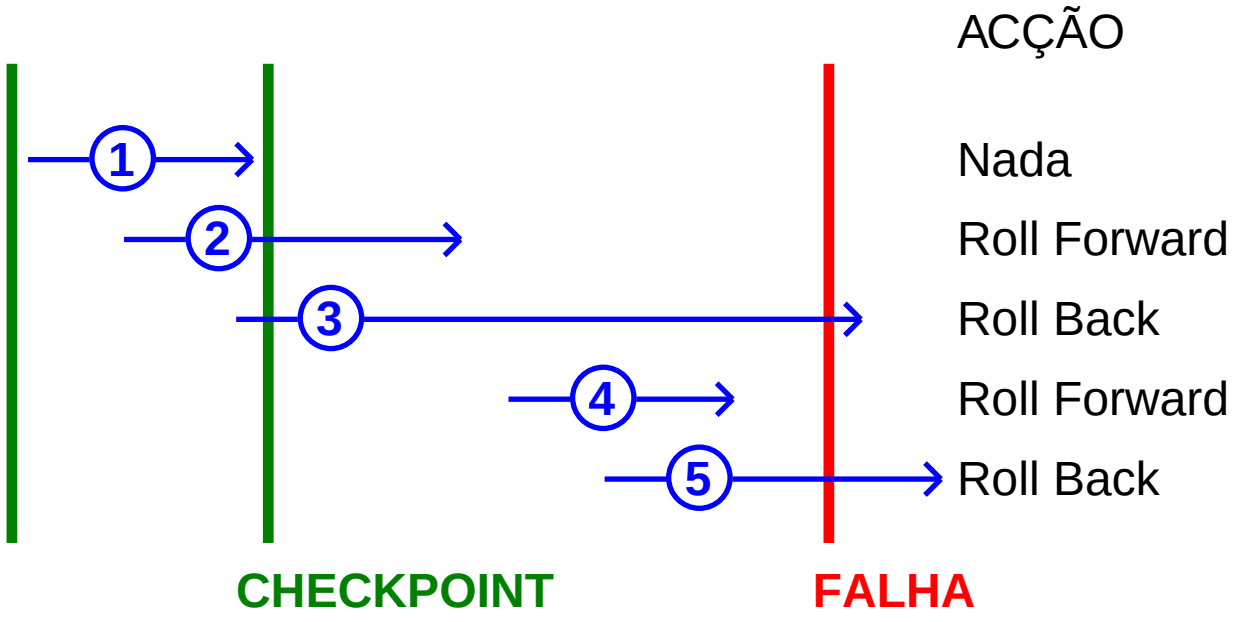
  O modo implícito termina com **`SET ... OFF`**

Quando o modo implícito termina - **`OFF`**, o SQL fica em modo Autocommit: as transacções ocorrem sem intervenção em cada operação feita na base. É o método predefinido no SQL Server e o que ocorre para cada instrução transmitida por ADO, OLE DB, ODBC ou outros. Cada transacção é *commited* ou *rolledback* quando a instrução termina.

Cada transacção é registada numa área designada por Transaction Log para manter a consistência da base de dados. Todas as alterações de dados são detectadas à medida que são executadas e cada modificação é registada antes de ser escrita na base de dados.

No Log cada operação fica registada. Ao início - **BEGIN** de uma transacção corresponde uma marca e o **COMMIT** tem outra de fim. O SQL Server efectua verificações – **CHECKPOINTS**, a tempos regulares e também eles são registados no Log de modo a identificar quais as transações que já foram aplicadas à base de dados. Quando ocorre um novo **CHECKPOINT** todas as transacções desde o anterior são escritas na base de dados. Isto liberta espaço no Log e acelera o tempo de recuperação. O processo de recuperação ocorre automaticamente sempre que o Server é re-iniciado depois de uma falha de energia ou qualquer outra falha do sistema Servidor ou Cliente ou por cancelamento a pedido.

Todas as transações *commited* são aplicadas à base de dados e as que não tiverem sido terminadas são **ROLLBACK**. O último **CHECKPOINT** é usado como ponto inicial de recuperação.



8.2. Considerações gerais

- As transacções devem ser pequenas.
- Não devem conter interacção com o utilizador. Se necessário deve ocorrer antes da transacção.
- Evitar usar transacções encadeadas: só as marcas exteriores são consideradas.
- Pode usar-se a variável global `@@trancount` para saber quantas transacções foram iniciadas.
- Algumas instruções SQL e algumas SP de sistema não podem ser usadas em transacções.

9. Bloqueios – LOCKS (SQL Server)

O mecanismo dos bloqueios previne os conflitos na actualização de dados - permite a concorrência. Os utilizadores não podem ler nem modificar dados que outros utilizadores estejam a modificar. Por exemplo num sistema de reserva de lugares, apenas uma pessoa consegue um lugar específico.

Situações que os *Locks* podem resolver:
- Perda de actualização: Dois utilizadores actualizam a mesma informação, mas só a última se reflecte na base de dados.
- Leitura suja: Um utilizador lê dados não *commited* por outra transacção.
- Leitura não repetível: Um documento é lido duas vezes mas entretanto foi modificado. A primeira leitura não se consegue repetir originando confusão.
- Fantasmas: Um utilizador actualiza uma colecção de registos, mas entretanto outro cria um nessa colecção. Ao ser lida pelo primeiro aparece um registo não actualizado.

Há duas especificações base que originam os dois métodos para assegurar a concorrência:

Controlo de concorrência Pessimista: Impede que mais do que uma aplicação aceda aos dados em simultâneo. É o mais usual em sistemas pesados.
Controlo de concorrência Optimista: As aplicações não bloqueiam os dados a que acedem. Se ocorre um conflito, uma das transacções terá de ser terminada e recomeçada depois.

9.1 Níveis de bloqueio - Granularidade

A granularidade - tipo de elemento a bloquear é determinada dinamicamente pelo SQL Server e está escalada desde o nível mais baixo: *row, page, key, key-range, index, ta*ble até ao mais alto: *database*.
Os objectos reais que são protegidos são:

Objecto	**Bloqueio efectuado**
RID	*Row Identifier* - Registo
Key	Índice
Page	Página de 8k de dados ou índices
Extent	Grupo de 8 páginas (dados ou índices)
Table	Uma tabela incluindo os índices
Database	Toda a base de dados

O SQL Server bloqueia os recursos automaticamente em função da tarefa.

9.2. *LOCK Mode*

O tipo de *lock* indica o nível de dependência que a conexão obtém do objecto bloqueado.

9.2.1. *Shared Lock (S)*

Permite que as transacções leiam os dados concorrentemente. Permite vários **`SELECT`** simultâneos mas não pode ocorrer um **`UPDATE`**.

9.2.2. *Update Lock (U)*

Uma única transacção pode pedir um *lock update* que será convertido em exclusivo. É uma espécie de *shared lock* com prioridade. Permite evitar um problema usual de dependência mútua de *locks*: uma situação designada por *deadlock* e que ocorre quando mais do que uma transacção obteve um *shared lock* e o quer promover. Se a transacção **T1** está pronta a actualizar os dados tenta promover o seu *lock* a um *lock* exclusivo. Mas se a transacção **T2** tem o mesmo *lock* e a mesma ideia de alteração, ambas ficam à espera que a outra liberte o seu *lock shared* para poderem promover o seu a exclusivo.

9.2.3. *Exclusive lock (X)*

É obtido acesso exclusivo ao objecto.

9.2.4. *Intent Lock*

É como se a transacção obtivesse um número de ordem de atendimento para obter um *lock* de um objecto. É uma espécie de reserva de posição para ser atendida. Os *intent lock* são feitos a tabelas e são de 3 tipos:

- *Intent Shared (IS)*: Indicam a intenção de ler
- *Intent Exclusive (IX)*: Indicam a intenção de alterar
- *Shared with Intent Exclusive (SIX)*: permitem que outros *intent locks* sejam colocados na tabela enquanto uma transacção têm a opção de efectuar um *lock* exclusivo.

9.2.5. *Schema Lock (Sch-M)*

É obtido por uma transacção que modifica a estrutura da base.

9.2.6. *Bulk Update Lock (BU)*

Usados em *bulk copy* ou *bulk insert*.

9.3. Factos

9.3.1. Duração dos *Locks*

Os *locks* são mantidos pelo tempo entendido como necessário para garantir a protecção dos dados.
Há vários cenários possíveis. Se o nível predefinido de isolamento de transacções estiver activo: *READ COMMITED*, então o *shared lock* é mantido o tempo necessário à leitura da página. Se a transacção tiver um *HOLD LOCK* o *lock* só é libertado depois da transacção terminar.

9.3.2. Protecção de *locks*

Nenhuma conexão pode obter um *lock* que conflitue com outro já concedido.

9.3.3. Compatibilidade

Só podem ser obtidos *locks* compatíveis com os já concedidos.

9.3.4. Escalabilidade

Os *locks* são automaticamente escalados do nível mínimo necessário à transacção para o máximo numa tentativa de diminuir os recursos usados. Quanto mais fino o *lock*, mais trabalho dá.

9.3.5. Informação

Pode obter-se informação sobre os locks com a *SP*: **`sp_lock`**

9.4. DEADLOCKS

Um *Deadlock* ocorre quando duas transacções tem *locks* em objectos separados e cada uma requer um *lock* no recurso da outra.

Os *deadlocks* podem ocorrer sem que seja possível ter um controlo sobre a situação pois podem resultar da forma como

por exemplo diferentes planos de execução de *queries* são executados.

Os *deadlocks* são terminados automaticamente pelo SQL Server cancelando uma das transacções e executando os seguintes passos:

1. **ROLLBACK** da transacção da vítima.
   (É a transacção que corre à menos tempo que é sacrificada)
2. Notifica a aplicação da vítima (erro 1205).
3. Cancela o pedido corrente da vítima.
4. Permite que a outra transacção continue.

Não sendo possível evitar os *deadlocks* devem observar-se as seguintes regras para reduzir o risco da sua ocorrência:

- Usar os recursos o mais tarde possível e libertá-los o mais cedo.
- Usar os recursos pela mesma ordem em todas as transacções.
- Evitar a interacção com o utilizador.
- Escrever transacções com poucos passos.
- Encurtar o tempo das transacções trabalhando sobre poucas linhas.
- Usar os *locks* predefinidos pois são aqueles com que o optimizador melhor trabalha.