Structured Query Language - SQL

- Tópicos:
  - Linguagem de definição de dados
  - Estrutura básica de perguntas em SQL
  - Operações com conjuntos
  - Funções de agregação
  - Junções
  - Valores nulos
  - Vistas e relações derivadas
  - Subconsultas
  - Modificações de bases de dados
- Bibliografia:
  - Capítulo 3 e Secções 4.1 e 4.2 do livro recomendado

Subconsultas na cláusula where

- O SQL também permite subconsultas na cláusula **where**
- Estas subconsultas permitem fazer:
  - fazer testes de pertença a conjuntos resultado de consultas
  - fazer comparações entre conjuntos resultado de consultas
  - calcular a cardinalidade de conjuntos
- A tradução deste tipo de consultas para álgebra relacional não é tão direta
  - Em geral, é possível caso a caso traduzir para expressões de álgebra relacional
  - Mas não existe uma tradução direta de cada tipo destas consultas para um operador da álgebra relacional

Pertença a conjunto (in)

- Listar todos os clientes que têm contas e empréstimos no banco.

```
select distinct customer_name
from borrower
where customer_name in (select customer_name
                        from depositor)
```

- Encontrar todos os clientes que têm empréstimos mas não possuem contas no banco

```
select distinct customer_name
from borrower
where customer_name not in (select customer_name
                            from depositor)
```

- Encontrar todas as agências de Lisboa ou de Almada

```
select *
from branch
where branch_city in ('Lisboa','Almada')
```

Consulta de exemplo

- Listar todos os clientes que têm uma conta e empréstimos na agência de Perryride

```
select distinct customer_name
from borrower inner join loan using (loan_number)
where branch_name = 'Perryridge'  and
        (branch_name, customer_name ) in
      (select branch_name, customer_name
       from depositor, account
       where depositor.account_number =
                        account.account_number )
```

- Nota: A consulta acima pode ser escrita de uma maneira muito mais simples. A formulação utilizada serve apenas para ilustrar as possibilidades da linguagem SQL.

Redundância do operador in

- Grande parte as vezes não é necessário usar o operador **in**
  - E é muito mais simples, não o usar!
- Listar todos os clientes que têm contas e empréstimos no banco.

```
select distinct customer_name
from borrower
where customer_name in (select customer_name
                        from depositor)
```

- Pode ser feito com

```
select distinct customer_name
from borrower inner join depositor using (customer_name)
```

Redundância do operador in

- Grande parte as vezes não é necessário usar o operador **in**
  - E é muito mais simples, não o usar!
- Encontrar todos os clientes que têm empréstimos mas não possuem contas no banco

  select distinct *customer_name*
  from *borrower*
  where *customer_name* **not in (select** *customer_name*
  from *depositor)*
- Pode ser feito com

  select distinct *customer_name*
  from *borrower* **left outer join** *depositor* **using** (*customer_name*)
  where *depositor.account_number* **is null**

Comparação de conjuntos (some)

- Apresentar todas as agências que têm activos superiores aos de alguma agência localizada em Brooklyn.

```
select distinct  T.branch_name
from branch as T, branch as S
where  T.assets > S.assets and
          S.branch_city = 'Brooklyn'
```

- A mesma consulta recorrendo à cláusula > **some**

```
select branch_name
from branch
where assets > some
          (select assets
           from branch
           where branch_city = 'Brooklyn')
```

Definição da cláusula Some

- F <comp> **some** *r* ⇔ ∃ *t* ∈ *r* : (F <comp> *t*)
  em que <comp> pode ser: <, >, ≤, ≥, =, ≠

(5< **some** | 0 / 5 / 6 |) = true
(ler: 5 menor que algum tuplo na relação)

(5< **some** | 0 / 5 |) = false

(5 = **some** | 0 / 5 |) = true

(5 ≠ **some** | 0 / 5 |) = true (pois 0 ≠ 5)

= **some** é o mesmo que **in**

No entanto, ≠ **some** não é o mesmo que **not in**

Cláusula all

- Listar os nomes das agências com activos superiores aos de todas as agências localizadas em Brooklyn.

```
select branch_name
from branch
where assets > all
        (select assets
        from branch
        where branch_city = 'Brooklyn')
```

- Sem o **all**

```
(select branch_name from branch)

        except

(select T.branch_name
 from branch T,branch S
 where S.branch_city = 'Brooklyn' and T.assets <=
S.assets)
```

Definição da cláusula all

- F <comp> **all** *r* ⇔ ∀ *t* ∈ *r* : (F <comp> *t*)
  em que <comp> pode ser: <, >, ≤, ≥, =, ≠
  - Se r for o conjunto vazio então F <comp> **all** *r* devolve **true**

(5< **all** | 0 / 5 / 6 |) = false

(5< **all** | 6 / 10 |) = true

(5 = **all** | 4 / 5 |) = false

(5 ≠ **all** | 4 / 6 |) = true (dado que 5 ≠ 4 e 5 ≠6)

(≠**all**) é o mesmo que **not in**

Contudo, (= **all**) não é o mesmo que **in**

Teste de Relações Vazias

- A construção **exists** devolve o valor **true** se a subconsulta é não vazia.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$
- Encontrar os clientes que têm uma conta e um empréstimo

```
select customer_name
    from borrower
    where exists (select *
            from depositor
            where depositor.customer_name = borrower.customer_name)
```

Cláusula "exists"

- Encontrar todas as cadeiras leccionadas no 1º semestre de 2016/17 e no 2º semestre de 2017/8:

```
select cod_cadeira
from cadeiras as S
where semestre = 1 and ano = 2016 and
        exists  (select *
                 from cadeiras as T
                 where semestre = 2 and ano = 2018
                       and S.cod_cadeira = T.cod_cadeira);
```

- **Nome de correlação** – variável S na consulta externa
- **Subconsulta correlacionada** – a consulta no interior

Cláusula contains

- Listar todos os clientes que têm conta em todas as agências de Brooklyn.

```
select distinct S.customer_name
from depositor as S
where
        (select R.branch_name
        from depositor as T, account as R
                where T.account_number = R.account_number and
                        S.customer_name = T.customer_name)
        contains
                (select branch_name
                 from branch
                 where branch_city = 'Brooklyn')
```

- ***Nota:*** Não existe no Oracle, o que não é grave pois:

$$X \supseteq Y \Leftrightarrow Y - X = \emptyset$$

Consulta de exemplo

- Listar todos os clientes que têm conta em todas as agências de Brooklyn.

```
select distinct S.customer_name
from depositor as S
where not exists (
        (select branch_name
        from branch
        where branch_city = 'Brooklyn')
                except
        (select R.branch_name
        from depositor as T, account as R
        where T.account_number = R.account_number and
                S.customer_name = T.customer_name ))
```

- ***Notas*:**
  - Repare que $X \supseteq Y \Leftrightarrow Y - X = \emptyset$
  - Não se pode escrever esta consulta com combinações de = **all** ou de suas variantes.
  - Em álgebra relacional esta consulta escrever-se-ia com uma divisão:

$$\Pi_{customer\_name,branch\_name}(\text{depositor} \bowtie \text{account}) \div \Pi_{branch\_name}(\sigma_{branch\_city='Brooklyn'}(\text{branch}))$$

Divisão em SQL

- $r \div s = \{ t \mid t \in \prod_{R-S}(r) \wedge \forall u \in s (tu \in r) \}$
- De forma equivalente:
  - Seja $q = r \div s$
  - Então q é a maior relação satisfazendo $q \times s \subseteq r$
- Seja r(A,B) e s(B). Em SQL, $r \div s$ é obtido por:

```
select distinct X.A
from r as X
where  (select Y.B from r as Y where X.A = Y.A)
       contains
       (select B from s)
```

- Ou então (o que já funciona no Oracle):

```
select distinct X.A from r as X
where  not exists ((select B from s)
                   except
                   (select Y.B from r as Y where X.A = Y.A)
                  )
```

Testar ausência de tuplos duplicados

- A construção **unique** verifica se o resultado de uma subconsulta possui tuplos duplicados.
- Encontrar todos os clientes que têm uma só conta na agência de Perryridge.

```
select T.customer_name
  from depositor as T
  where unique (
        select R.customer_name
        from account, depositor as R
        where T.customer_name = R.customer_name and
              R.account_number = account.account_number and
              account.branch_name = 'Perryridge')
```

- Esta construção não está disponível no Oracle
  - O que também não é grave pois:

unique (select As **from** ...**)**

é equivalente a

1 = **all (select count(**As) **from** ... **group by** As**)**

Consulta de exemplo

- Listar todos os clientes que têm pelo menos duas contas na agência de Perryridge.

```
select distinct T.customer_name
from depositor as T
where not unique (
      select R.customer_name
      from account, depositor as R
      where T.customer_name = R.customer_name and
        R.account_number = account.account_number and
        account.branch_name = 'Perryridge')
```

- Ou então (de forma bem mais simples!):

```
select customer_name
from depositor inner join account using (account_number)
where branch_name = 'Perryridge'
group by customer_name
having count(distinct account_number) > 1
```

Subconsulta escalar

- Uma subconsulta escalar é aquela em que apenas se espera obter um único valor
- Listar todos os departamentos juntamente com o número de docentes de cada departamento

```
select dept_name,
        ( select count(*)
          from instructor
          where department.dept_name = instructor.dept_name)
        as num_instructors
from department;
```

- Ocorre um erro em tempo de execução se a subconsulta devolver mais do que um tuplo

Modificação da base de Dados – Remoção

- A remoção de tuplos de uma tabela (ou vista) é feita em SQL com a instrução

  delete from *<tabela ou vista>*
  where *<Condição>*

- Apagar todas as contas da agência de Perryridge

  delete from *account*
  where *branch-name* = 'Perryridge'

Exemplo de remoção

- Apagar todas as contas de todas as agências na cidade de Needham.

```
delete from depositor
where account-number in
                (select account-number
                  from branch natural inner join account
                  where branch-city = 'Needham')

delete from account
where branch-name in (select branch-name
                              from branch
                              where branch-city = 'Needham')
```

Exemplo de remoção

- Remover todas as contas com saldos inferiores aos da média do banco.

  delete from *account*
  where *balance* < **some** (**select avg** *(balance)*
  from *account)*

  - Problema:
    - à medida que removemos tuplos de *account,* o saldo médio altera-se
  - Solução utilizada no standard SQL:
    1. Primeiro, calcula-se o saldo médio (**avg**) e determinam-se quais os tuplos a apagar
    2. Seguidamente, removem-se todos os tuplos identificados anteriormente (sem recalcular **avg** ou testar novamento os tuplos)

Modificação da base de dados – Inserção

- A inserção de tuplos numa tabela (ou vista) é feita em SQL com a instrução

  insert into *<tabela ou vista>*
  values *<Conjunto de tuplos>*

ou

  insert into *<tabela ou vista>*
  select *...*

- Adicionar um novo tuplo a *account*

  insert into *account*
  values ('A-9732', 'Perryridge',1200)

  ou equivalentemente

  insert into *account (branch-name, balance, account-number)*
  values ('Perryridge', 1200, 'A-9732')

- Adicionar um novo tuplo a *account* em que *balance* é null

  insert into *account*
  values ('A-777','Perryridge', *null*)

Exemplo de Inserção

- Dar como bónus a todos os mutuários da agência de Perryride, uma conta de poupança de €200. O número do empréstimo servirá de número de conta de poupança

```
insert into account
    select loan-number, branch-name, 200
    from loan
    where branch-name = 'Perryridge'
insert into depositor
    select customer-name, loan-number
    from loan natural inner join borrower
    where branch-name = 'Perryridge'
```

- A instrução **select-from-where** é avaliada antes da inserção de tuplos na relação (caso contrário consultas como

```
insert into table1 select * from table1
```

causariam problemas)

Modificação da base de dados – Actualização

- A actualização de tuplos duma tabela (ou vista) é feita em SQL com a instrução

  update *<tabela ou vista>*
  set *<Atributo> = <Expressão>,*
  <Atributo> = <Expressão>, ...
  where *<Condição>*

- Pagar juros de 1% a todas as contas da agência Perryride.

  update *account*
  set *balance = balance* * 1.01
  where *branch_name =* 'Perryride'

Modificação da base de dados – Actualização

- Pagar juros de 6% a todas as contas com saldos superiores a €10,000, e juros de 5% às restantes contas.
  - Escrever duas instruções de **update**:

update *account*
set *balance = balance* * 1.06
where *balance* > 10000

update *account*
set *balance = balance* * 1.05
where *balance* ≤ 10000

  - A ordem é importante!
    - Porquê?
  - Pode ser feito de forma mais "limpa" recorrendo à instrução **case**

Instrução Case para Actualizações Condicionais

- Pagar juros de 6% a todas as contas com saldos superiores a €10,000, e juros de 5% às restantes contas.

```
update account
set balance =  case
                  when balance <= 10000 then balance *1.05
                  else   balance * 1.06
               end
```

Atualizações com subconsultas escalares

- Atualizar o total de créditos para todos os estudantes

```
update alunos S
set tot_cred = (select sum(creditos)
                from inscricoes inner join cadeiras using(cod_cadeira)
                 where alunos.numero = inscricoes.numero and
                       inscritos.nota <> 'R' and
                       inscritos.nota is not null );
```

- Coloca *tot_creds* a null para estudantes que não tenham realizado qualquer cadeira
- Em vez de **sum**(*credits*), escrever:

```
case
   when sum(creditos) is not null then sum(creditos)
   else 0
end
```

Atualização de uma vista

- Modificações nas bases de dados através de vistas devem ser traduzidas para modificações das verdadeiras relações presentes na base de dados.
  - E.g com uma vista com a informação sobre empréstimos, escondendo o atributo *amount*

    create view *branch-loan* **as**
    select *branch-name, loan-number*
    from *loan*

  - A adição de um novo tuplo em *branch-loan*

    insert into *branch-loan* **values** (‘Perryridge’, ‘L-307’)

    causa problemas pois terá que ser traduzido em adições de tuplos em tabelas que existam na base de dados.
  - Duas hipóteses:
    - Rejeitar a inserção e devolver uma mensagem de erro
    - Traduzir na inserção, na relação *loan,* do tuplo

      (‘L-307’, ‘Perryridge’, *null*)

Atualização de uma vista (cont.)

- Outro problema ocorre quando temos, por exemplo, a vista:

  create view *info_empréstimos* **as**
  select *customer_name, amount*
  from *borrower* **natural inner join** *loan*

e pretendemos fazer a seguinte inserção:

insert into *info_empréstimos* **values** *('Johnson',1900)*

- *A única forma seria inserir ('Johnson' ,null) na tabela* borrower *e (null,null,1900) na tabela* loan, não tendo o efeito desejado.

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500
null	*null*	1900

loan

customer_name	*loan_number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17
Johnson	*null*

borrower

Atualização de uma vista (cont)

- Outras não têm tradução única, como por exemplo:

  create view *all_costumers* **as**
  (select * **from** *depositor)*

  union
  *(***select** * **from** *borrower)*

- Toda a adição em *all_costumers* não tem tradução única:
  - Deve introduzir-se em *depositor* ou em *borrower*???

Atualização de vistas

- Uma view em SQL é atualizável (updatable) se todas as seguintes condições se verificam:
  - A clausula **from** só contém uma relação da base de dados;
  - A clausula **select** apenas contém nomes de atributos da relação, não contendo expressões, agregados, ou especificação de **distinct**;
  - Qualquer atributo que não aparece na clausula **select** deve poder tomar o valor null;
  - A consulta não contém nenhuma clausula **group by** nem **having**.
- A view

  create view *downtown_account* **as**
  select *acount_number,branch_name,balance*
  from *account*
  where *branch-name* = 'Downtown'

... é atualizável. No entanto, a inserção

insert into *downtown_account* **values** ('L-307','Perryridge',1000)

apesar de ser efetuada, não produziria efeitos na view.

Atualização de vistas

- Em Oracle é possível impedir as situações anteriores por intermédio da cláusula WITH CHECK OPTION na criação da vista.

  create view *downtown_account* **as**
  select *acount_number,branch_name,balance*
  from *account*
  where *branch-name* = 'Downtown'
  with check option

- Para impedir a atualização de vistas utiliza-se a cláusula WITH READ ONLY

Atualização de vistas com joins

- Em Oracle é possível em certas circunstâncias efetuar atualizações de vistas definidas com joins
- Como regra genérica, as modificações só podem alterar uma única das tabelas da vista
- Para o caso das inserções recorre ao conceito de 'Key-Preserved Tables'

Integração de SQL com outras linguagens de programação

- Tópicos:
  - Embedded e Dynamic SQL
  - Ligações por ODBC e JDBC
  - Linguagens proprietárias
- Bibliografia:
  - Secções 5.1 e 5.2 do livro recomendado

SQL a partir de uma linguagem de programação

- SQL fornece uma linguagem *declarativa* para manipulação de bases de dados. Facilita a manipulação e permite optimizações muito difíceis se fossem programadas em linguagens imperativas.

- Mas há razões para usar SQL juntamente com linguagens de programação gerais (imperativas):
  - o SQL não tem a expressividade de uma máquina de Turing (há perguntas impossíveis de codificar em SQL – e.g. fechos transitivos)
    - usando SQL juntamente com linguagens gerais é possível suprir esta deficiência
  - nem tudo nas aplicações de bases de dados é declarativo (e.g. ações de afixar resultados, interfaces, etc)
    - Essa parte pode ser programado em linguagens gerais

SQL a partir de uma linguagem de programação

Há duas aproximações possíveis para aceder a SQL a partir de uma linguagem de programação genérica

- Um programa pode ligar-se e comunicar com um servidor de bases de dados utilizando um conjunto de funções de uma API
- Embedded SQL – disponibiliza um meio de colocar código diretamente no programa. Os comandos SQL são traduzidos em tempo de compilação para chamadas de funções. Em tempo de execução estas chamadas de funções ligam-se à base de dados recorrendo a uma API que disponibiliza capacidades de SQL dinâmico.
- O standard SQL define uma série de embeddings, para várias linguagens de programação (Pascal, PL/I, C, C++, Java, etc).
  - À linguagem na qual se incluem comandos SQL chama-se linguagem *host*. Às estruturas SQL permitidas na linguagem *host* chama-se SQL embutido (ou *embedded* SQL)

JDBC

- JDBC é uma API **Java** para comunicar com sistemas de bases de dados que suportam o SQL.
- JDBC suporta várias formas de consulta e modificação de bases de dados
- JDBC também disponibiliza formas de obter meta-informação sobre a base de dados, nomeadamente que tabelas existem e seus atributos
- O modelo de comunicação com a base de dados:
  - Abre uma ligação
  - Cria um objecto “statement”
  - Executa comandos usando esse objecto para enviar os comandos e obter os resultados
  - Usa mecanismos de excepção para lidar com os erros

Esqueleto de código JDBC

```
public static void JDBCexample(String dbid, String userid, String passwd)
  {
   try (Connection conn = DriverManager.getConnection(
         "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
         Statement stmt = conn.createStatement();
       )
   {
         … Do Actual Work ….
   }
   catch (SQLException sqle) {
      System.out.println("SQLException : " + sqle);
  }
}
```

NOTA: Código anteriores funcionam para versões de Java 7 e JDBC 4 e posteriores. Recursos abertos no bloco “try (….)” (“try with resources”) são automaticamente encerrados no final do try

Código JDBC para versões antigas Java/JDBC

```
public static void JDBCexample(String dbid, String userid, String passwd)
 {
   try {
      Class.forName ("oracle.jdbc.driver.OracleDriver");
      Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
      Statement stmt = conn.createStatement();
         … Do Actual Work ….
      stmt.close();
      conn.close();
   }
   catch (SQLException sqle) {
      System.out.println("SQLException : " + sqle);
   }
}
```

NOTA: Classs.forName não é necessário a partir da versão JDBC 4 onwards. A sintaxe "try with resources" no slide anterior é preferível para Java 7/8.

Exemplo de código JDBC (Cont.)

- Actualização

```
try {
    stmt.executeUpdate(  "insert into account values
                                ('A-9732', 'Perryridge', 1200)");
} catch (SQLException sqle) {
    System.out.println("Could not insert tuple. " + sqle);
}
```

- Execução de perguntas

```
ResultSet rset = stmt.executeQuery( "select branch_name, avg(balance)
                                     from account
                                     group by branch_name");
while (rset.next()) {
    System.out.println(
            rset.getString("branch_name") + "  " + rset.getFloat(2));

}
```

Funcionalidades JDBC

- Ligação à Base de Dados
- Envio de comandos SQL para o SGBD
- Excepções e Gestão de Recursos
- Obter os resultados de uma consulta
- Comandos preparados
- Chamada de funções e procedimentos
- Metainformação

Detalhes de código JDBC

- Obter resultados de campos:
  - **rs.getString(“dept_name”) e rs.getString(1) são equivalentes se dept_name é a primeira coluna no comando select executado.**
- Tratatamento de valores nulos

```
int a = rs.getInt(“a”);
if (rs.wasNull()) Systems.out.println(“Got null value”);
```

Comandos Preparados

- PreparedStatement pStmt = conn.prepareStatement(
  "insert into instructor values(?,?,?,?)");
  pStmt.setString(1, "88877");
  pStmt.setString(2, "Perry");
  pStmt.setString(3, "Finance");
  pStmt.setInt(4, 125000);
  pStmt.executeUpdate();
  pStmt.setString(1, "88878");
  pStmt.executeUpdate();
- **AVISO**: utilizar **sempre** comandos preparados quando se utilizar input do utilizador e adicioná-lo a uma query
  - NUNCA criar a consulta por concatenação de strings
  - "insert into instructor values(' " + ID + " ', ' " + name + " ', " + " ' + dept name + " ', " ' balance + ')“
  - Que acontece se o nome for “D'Souza”?

Injeção de código SQL

- Suponha que a consulta é construída com
  - "select * from instructor where name = '" + name + "'"
- Suponha que o utilizador, em vez de introduzir um nome, escreve:
  - X' or 'Y' = 'Y
- o comando resultante fica:
  - "select * from instructor where name = '" + "X' or 'Y' = 'Y" + "'"
  - ou seja:
    - select * from instructor where name = 'X' or 'Y' = 'Y'
  - Utilizador poderia ter colocado antes…
    - X'; update instructor set salary = salary + 10000; --
- O comando preparado internamente traduz para:
  "select * from instructor where name = 'X\' or \'Y\' = \'Y'
  - **Utilizar sempre comandos preparados com input do utilizador como parâmetros**

Metainformação

- Metainformação sobre o ResultSet
- Por exemplo após execução de uma consulta para obter um ResultSet rs:
  - ResultSetMetaData rsmd = rs.getMetaData();

```
for(int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i));
    System.out.println(rsmd.getColumnTypeName(i));
}
```

- Como pode ser útil?

Metainformação (Cont)

- Metainformação sobre a base de dados
- DatabaseMetaData dbmd = conn.getMetaData();

```
// Arguments to getColumns: Catalog, Schema-pattern, Table-pattern,
// and Column-Pattern
// Returns: One row for each column; row has a number of attributes
// such as COLUMN_NAME, TYPE_NAME
// The value null indicates all Catalogs/Schemas.
// The value “” indicates current catalog/schema
// The value “%” has the same meaning as SQL like clause
ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
while( rs.next()) {
      System.out.println(rs.getString("COLUMN_NAME"),
                         rs.getString("TYPE_NAME");
}
```

- Como pode ser útil?

Metadata (Cont)

- Metainformação sobre a base de dados
- DatabaseMetaData dbmd = conn.getMetaData();

```
// Arguments to getTables: Catalog, Schema-pattern, Table-pattern,
// and Table-Type
// Returns: One row for each table; row has a number of attributes
// such as TABLE_NAME, TABLE_CAT, TABLE_TYPE, ..
// The value null indicates all Catalogs/Schemas.
// The value “” indicates current catalog/schema
// The value “%” has the same meaning as SQL like clause
// The last attribute is an array of types of tables to return.
// TABLE means only regular tables

ResultSet rs = dbmd.getTables (“”, "", “%", new String[] {“TABLES”});

while( rs.next()) {

      System.out.println(rs.getString(“TABLE_NAME“));

}
```

Encontrar as chaves primárias

- DatabaseMetaData dmd = connection.getMetaData();

```
// Arguments below are:  Catalog, Schema, and Table
// The value “”  for Catalog/Schema indicates current catalog/
schema
//  The value null indicates all catalogs/schemas
ResultSet rs = dmd.getPrimaryKeys(“”, “”, tableName);

while(rs.next()){
    // KEY_SEQ indicates the position of the attribute in
    // the primary key, which is required if a primary key has multiple
    // attributes
    System.out.println(rs.getString(“KEY_SEQ”),
                                rs.getString("COLUMN_NAME");
}
```

Transaction Control in JDBC

- By default, each SQL statement is treated as a separate transaction that is committed automatically
  - bad idea for transactions with multiple updates
- Can turn off automatic commit on a connection
  - conn.setAutoCommit(false);
- Transactions must then be committed or rolled back explicitly
  - conn.commit(); or
  - conn.rollback();
- conn.setAutoCommit(true) turns on automatic commit.

Outras capacidades do JDBC

- Chamar funções e procedimentos
  - CallableStatement cStmt1 = conn.prepareCall("{? = call some function(?)}");
  - CallableStatement cStmt2 = conn.prepareCall("{call some procedure(?,?)}");
- Lidar com grandes objectos
  - getBlob() e getClob() são semelhantes ao método getString(), mas devolvem objetos do tipo Blob e Clob, respetivamente
  - Obter os dados com getBytes()
  - Associar uma stream previamente aberta com o objeto Java Blob ou Clob para atualizar objetos grandes
    - blob.setBlob(int parameterIndex, InputStream inputStream).

Para saber mais sobre JDBC

- JDBC Basics Tutorial
  - https://docs.oracle.com/javase/tutorial/jdbc/index.html

Embedded SQL

- JDBC é demasiado dinâmico, erros não podem ser detetados pelo compilador
- O SQL embutido permite acesso a bases de dados, via outra linguagens de programação.
  - Toda a parte de acesso e manipulação da base de dados é feito através de código embutido. Todo o processamento associado é feito pelo sistema de bases de dados. A linguagem *host* recebe os resultados e manipula-os.
  - O código tem que ser pré-processado. A parte SQL é transformada em código da linguagem *host,* mais chamadas a run-time do servidor.
- A expressão EXEC SQL é usado para identificar código SQL embutido

  EXEC SQL <embedded SQL statement > END-EXEC

  Nota: Este formato varia de linguagem para linguagem. E.g. em C usa-se ';' em vez do END-EXEC.

  Em Java usa-se # SQL { …. } ;

SQLJ

- SQLJ: SQL embutido em Java
  - #sql iterator deptInfoIter (String dept name, int avgSal);

```
deptInfoIter iter = null;
#sql iter = { select dept_name, avg(salary) from instructor
            group by dept name };
while (iter.next()) {
    String deptName = iter.dept_name();
     int avgSal = iter.avgSal();
     System.out.println(deptName + " " + avgSal);
}
iter.close();
```

Cursores

- Para executar um comando SQL numa linguagem *host* é necessário começar por declarar um cursor para esse comando.
- O comando pode conter variáveis da linguagem *host,* precedidas de :
- E.g. Encontrar os nome e cidades de clientes cujo saldo seja superior a *amount*

EXEC SQL

declare *c* **cursor for**
select *customer-name, customer-city*
from *account* **natural inner join** *depositor*
natural inner join *customer*
where *account.balance > :amount*

END-EXEC

Embedded SQL (Cont.)

- O comando **open** inicia a avaliação da consulta no cursor

  EXEC SQL **open** *c* END-EXEC

- O comando **fetch** coloca o valor de um tuplo em variáveis da linguagem *host*.

  EXEC SQL **fetch** *c* **into** :*cn, :cc* END-EXEC

- Chamadas sucessivas a **fetch** obtêm tuplos sucessivos
- Uma variável chamada SQLSTATE na *SQL communication area* (SQLCA) toma o valor '02000' quando não há mais dados.
- O comando **close** apaga a relação temporária, criada pelo **open**, que contem os resultados da avaliação do SQL.

  EXEC SQL **close** *c* END-EXEC

Modificações com Cursores

- Como não devolvem resultado, o tratamento de modificações dentro doutras linguagens é mais fácil.
- Basta chamar qualquer comando válido SQL de **insert, delete,** ou **update** entre EXEC SQL e END SQL

- Em geral, as váriaveis da linguagem *host* só podem ser usadas em locais onde se poderiam colocar variáveis SQL.
- Não é possível *construir* comandos (ou parte deles) manipulando strings da linguagem *host*

Dynamic SQL

- Permite construir e (mandar) executar comandos SQL, em run-time.
- E.g. (chamando dynamic SQL, dentro de um programa em C)

```
char *  sqlprog = "update account
                   set balance = balance * 1.05
                   where account-number = ?";
EXEC SQL prepare dynprog  from :sqlprog;
char account [10] = "A-101";
EXEC SQL execute dynprog using :account;
```

- A string contém um ?, que indica o local onde colocar o valor a ser passado no momento da chamada para execução.

ODBC

- Standard Open DataBase Connectivity(ODBC)
  - Standard para comunicação entre programas e servidores de bases de dados
  - application program interface (API) para
    - Abrir uma ligação a uma base de dados
    - Enviar consultas e pedidos de modificações
    - Obter os resultados
- Aplicações diversas (e.g. GUI, spreadsheets, etc) podem usar ODBC