RAWDATA
Section 1

SQL part 3
SQL & Programming

Henrik Bulskov & Troels Andreasen

SQL & Programming

- Accessing SQL From a Programming Language
  - JDBC, ODBC and ADO.NET
    - ADO.NET with C# will be covered in more detail in section two

- Programming the database
  - Functions and Procedural Constructs in SQL
  - Triggers in SQL
  - (Scheduled events in MySQL)

JDBC and ODBC and ADO.NET

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
  - Connect with the database server
  - Send SQL commands to the database server
  - Fetch tuples of result one-by-one into program variables

- JDBC (Java Database Connectivity)
  - works with Java
- ODBC (Open Database Connectivity)
  - works with C, C++, C#, and Visual Basic
- ADO.NET
  - works with the .NET framwwork
  - will be used with C# on RAWDATA
    - especially with what's called Entity-Framework and LINQ
  -

JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL.
- JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- Model for communicating with the database:
  1) Open a connection
  2) Create a “statement” object
  3) Execute queries using the Statement object to send queries and fetch results
  4) Extract data from result set
  5) Close connection
  – (Use exception mechanism to handle errors)

JDBC Code

```
import java.sql.*;

public class FirstExample {
   static final String DB_URL = "jdbc:mysql://" + "localhost:3306" +  "/" + "university"; // a JDBC url
   static final String USER = "troels";
   static final String PASS = ”xxxx";

   public static void main(String[] args) {
   Connection conn = null;
   Statement stmt = null;
   try{
      Class.forName("com.mysql.jdbc.Driver");

      // Open a connection
      System.out.println("Connecting to database...");
      conn = DriverManager.getConnection(DB_URL,USER,PASS);

      // Create a statement
      System.out.println("Creating statement...");
      stmt = conn.createStatement();
      String sql;
      sql = "SELECT id, name, salary FROM instructor";
      // Execute a query
      ResultSet rs = stmt.executeQuery(sql);

      // Extract data from result set
      while(rs.next()){
         //Retrieve by column name
         String id = rs.getString("id");
         String name = rs.getString("name");
         int salary = rs.getInt("salary");

         //Display values
         System.out.print("ID: " + id);
         System.out.print(", Name: " + name);
         System.out.println(", Salary: " + salary);
      }
      // Close Connection
      rs.close();
      stmt.close();
      conn.close();
   }catch(SQLException se){
         //Handle errors for JDBC
         se.printStackTrace();
      }catch(Exception e){
         //Handle errors for Class.forName
         e.printStackTrace();
      }finally{
         //finally block used to close resources
         try{
            if(stmt!=null)
               stmt.close();
         }catch(SQLException se2){
         }// nothing we can do
         try{
            if(conn!=null)
               conn.close();
         }catch(SQLException se){
            se.printStackTrace();
         }//end finally try
      }//end try
      System.out.println("Goodbye!");
   }//end main
   }//end FirstExample
```

```
> run FirstExample
Connecting to database...
Creating statement...
ID: 10101, Name: Srinivasan, Salary: 65000
ID: 12121, Name: Wu, Salary: 90000
ID: 15151, Name: Mozart, Salary: 40000
ID: 22222, Name: Einstein, Salary: 95000
ID: 30765, Name: Green, Salary: 0
ID: 30766, Name: Green, Salary: 0
ID: 32343, Name: El Said, Salary: 60000
ID: 33456, Name: Gold, Salary: 87000
ID: 45565, Name: Katz, Salary: 75000
ID: 58583, Name: Califieri, Salary: 62000
ID: 76543, Name: Singh, Salary: 80000
ID: 76766, Name: Crick, Salary: 72000
ID: 83821, Name: Brandt, Salary: 92000
ID: 98345, Name: Kim, Salary: 80000
Goodbye!
>
```

ADO.NET

- The ADO.NET API provides functions to access data similar to the JDBC functions.
- Thus ADO.NET allows access to results of SQL queries
- A similar model for communicating with the database:
  1) Open a connection
  2) Create a “statement” object
  3) Execute queries using the Statement object to send queries and fetch results
  4) Extract data from result set
  5) Close connection

ADO.NET

```
ID: 10101, Name: Srinivasan, Salary: 65000
ID: 12121, Name: Wu, Salary: 90000
ID: 15151, Name: Mozart, Salary: 40000
ID: 22222, Name: Einstein, Salary: 95000
ID: 32343, Name: El Said, Salary: 60000
ID: 33456, Name: Gold, Salary: 87000
ID: 45565, Name: Katz, Salary: 75000
ID: 58583, Name: Califieri, Salary: 62000
ID: 76543, Name: Singh, Salary: 80000
ID: 76766, Name: Crick, Salary: 72000
ID: 83821, Name: Brandt, Salary: 92000
ID: 98345, Name: Kim, Salary: 80000

Press any key to continue...
```

```
using MySql.Data.MySqlClient;
using System;

namespace ADOExample
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var conn = new MySqlConnection(
                "server=wt-220.ruc.dk;database=university;uid=troels;pwd=xxxx;SslMode=none"))
            {
                conn.Open();
                var cmd = new MySqlCommand("SELECT id, name, salary FROM university.instructor",conn);
                var rdr = cmd.ExecuteReader();

                while(rdr.Read())
                {
                    Console.WriteLine($"{"ID: " + rdr.GetInt32(0)}{", Name: " + rdr.GetString(1)}{", Salary: "+rdr.GetInt32(2)}");
                }
                rdr.Close();
                conn.Close();
            }
        }
    }
}
```

Functions and Procedural Constructs in SQL

Procedural Extensions and Stored Procedures

- SQL provides a **module** language
  - Permits definition of procedures in SQL

- Functions
  - write your own functions and add them to the database
  - can be used e.g. in select and where like any predefined function
- Stored Procedures
  - you can store procedures in the database
  - then execute them using the **call** statement
  - permit external applications to operate on the database without knowing about internal details
  - you can make your own dedicated API that provides functionality but hides the database structure
- Triggers
  - you can add special procedures that are executed automatically by the system as a side effect of a modification to the database

Functions and Procedures

- Since SQL:1999 the standard supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external programming language.
- SQL:1999 also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment, and others
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999.
- MySQL follows the SQL:2003 syntax for stored functions and procedures

SQL Functions

- Define a function.

```
create function hello (s char(20))
 returns char(50)
 begin
 return concat('hello, ',s,'!');
 end;
```

- Use the function.

```
select hello('world') 'Message to all';
```

notice temporary change of delimiter

```
drop function if exists hello;

delimiter //
create function hello (s char(20))
returns char(50)
begin
return concat('hello, ',s,'!');
end;//
delimiter ;
```

```
mysql> select hello('world') 'Message t
+----------------+
| Message to all |
+----------------+
| hello, world!  |
+----------------+
1 row in set (0.00 sec)

mys
```

```
mysql> select hello(name) 'Message to all' from instructor;
+--------------------+
| Message to all     |
+--------------------+
| hello, Srinivasan! |
| hello, Wu!         |
| hello, Mozart!     |
```

Semicolon is default a statement delimiter. You must redefine the delimiter temporarily to to pass the entire stored program definition to the server.

SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
returns integer
begin
    declare d_count integer;
    select count (* ) into d_count
    from instructor
    where instructor.dept_name = dept_name
    return d_count;
end
```

- Find the department name and budget of all departments with more that 1 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name ) > 1
```

SQL Functions

- Same function, defined in MySQL (notice temporary change of delimiter)
- Again count of the number of instructors in that department.

```
drop function if exists dept_count;

delimiter //
create function dept_count (dept_name char(20))
returns integer
begin
    declare d_count integer;
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dept_name;
    return d_count;
end;//
delimiter ;
```

```
mysql> select
       dept_count('Physics');
+-----------------------+
| dept_count('Physics') |
+-----------------------+
|                     2 |
+-----------------------+
1 row in set (0.00 sec)
```

```
mysql> select dept_name, budget from depart
    -> where dept_count (dept_name ) > 1;
+------------+-----------+
| dept_name  | budget    |
+------------+-----------+
| Comp. Sci. | 100000.00 |
| Finance    | 120000.00 |
| History    |  50000.00 |
| Physics    |  70000.00 |
+------------+-----------+
```

SQL Procedures

- The *dept_count* function could instead be written as procedure:

```
create procedure dept_count_proc(in dept_name varchar(20),
                                 out d_count integer)
begin
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dept_name;
end
```

- Procedures can be called , using the **call** statement, from
  - other procedures or
  - SQL embedded in application programs or
  - command line.

SQL Procedures

```
drop procedure if exists dept_count_proc;

delimiter //
create procedure dept_count_proc (in dept_name varchar(20),
                                  out d_count integer)
begin
   select count(*) into d_count
   from instructor
   where instructor.dept_name = dept_name;
end;//
delimiter ;
```

calling the procedure from the command line

```
mysql> CALL dept_count_proc('Physics', @out_value);
Query OK, 1 row affected (0.00 sec)

mysql> Select @out_value;
+-------------+
| @out_value |
+-------------+
|           2 |
+-------------+
1 row in set (0.00 sec)
```

MySQL variables

- So what was this: `@out_value`?

- **A User-Defined Variable** (See MySQL ref manual sec 9.4)
  - You can store a value in a user-defined variable in one statement and then refer to it later in another statement.
  - This enables you to pass values from one statement to another.
  - User variables are written as `@var_name`

- **User-Defined Variables, example**

```
mysql> SET @t1=1, @t2=2, @t3:=4;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @t1, @t2, @t3, @t4 := @t1+@t2+@t3;
+------+------+------+--------------------+
| @t1  | @t2  | @t3  | @t4 := @t1+@t2+@t3 |
+------+------+------+--------------------+
|    1 |    2 |    4 |                  7 |
+------+------+------+--------------------+
1 row in set (0.00 sec)
```

Procedural Constructs

- Conditional statements (**if-then-else**)

- Compound statement: **begin ... end**,
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within a compound statement

- Loops: **While** and **repeat** statements :

```
declare n integer default 0;
while n < 10 do
        set n = n + 1;
end while;

repeat
        set n = n − 1;
until n = 0
end repeat;
```

- Warning: most database systems implement their own variant of a modular (procedural) language – only inspired by the standard syntax

SQL Procedure, example (WHILE and Transaction)

```
drop table if exists foo;
create table foo
(
id int auto_increment primary key,
val numeric(6,0) not null default 0
);

drop procedure if exists load_foo;

delimiter //
create procedure load_foo()
begin
  declare i_max int default 4;
  declare i int default 0;

  start transaction;
  while i < i_max do
    insert into foo (val) values ((rand() * 10000));
    set i=i+1;
  end while;
  commit;
end //

delimiter ;
```

```
mysql> call load_foo();
Query OK, 0 rows affected, 1 warning

mysql> select * from foo order by id
+----+------+
| id | val  |
+----+------+
|  1 | 5406 |
|  2 | 8589 |
|  3 | 6725 |
|  4 | 7858 |
+----+------+
```

SQL Procedure, example (cont.)

- Notice SQL-details
  - drop … if exists …
    - `drop table if exists foo;`
    - `drop procedure if exists load_foo;`
  - auto_increment primary key
    - `id int auto_increment primary key,`
  - Declaration and initialization of variable
    - `declare i_max int default 4;`
  - while loop to do several DML-statements
    - `while i < i_max do`
      …
  - Transaction
    - `start transaction;`
      …
      `commit;`

Cursor

- **cursor**
  - is a control structure that enables traversal of rows in a table
  - a cursor is declared by a query and the table to be traversed is the result of this query
- **declare**
  - Before a cursor can be used it must be declared (defined).
  - `declare cur1 cursor for select name,salary from instructor;`
- **open** – perform the query
  - The cursor must be opened for use. This process actually retrieves the data using the previously defined SELECT statement.
  - `open cur1;`
- **fetch** – get the next row from the table
  - Individual rows can be fetched (retrieved) as needed.
  - `fetch cur1 into a, b;`
- **close** – close the cursor (clean up)
  - When done, the cursor must be closed.
  - `close cur1;`

SQL Procedure using cursor, example

```
delimiter //
create procedure curdemo()
begin
  declare done int default false;
  declare a char(16);
  declare b int;
  declare cur1 cursor for select name,salary from instructor;
  declare continue handler for not found set done = true;

  open cur1;

  read_loop: loop
    fetch cur1 into a, b;
    if done then
      leave read_loop;
    end if;
    if b > 81000 then
      insert into test values (a,b);
    end if;
  end loop;

  close cur1;
end;//
delimiter ;
```

```
mysql> truncate table test;
Query OK, 0 rows affected (0.01 sec)

mysql> call curdemo();
Query OK, 0 rows affected (0.01 sec)

mysql> select * from test;
+----------+----------+
| name     | salary   |
+----------+----------+
| Wu       | 90000.00 |
| Einstein | 95000.00 |
| Gold     | 87000.00 |
| Brandt   | 92000.00 |
+----------+----------+
4 rows in set (0.00 sec)
```

SQL Procedure using cursor, example(cont.)

- Notice SQL-details
  - Another loop construction
    - `loop … end loop;`
  - Label
    - `read_loop: loop …`

    permitted for BEGIN ... END blocks, LOOP, REPEAT, and WHILE
  - Label-reference,
    respectively jump to beginning, jump out of block:
    - `iterate read_loop;`
    - `leave read_loop;`
  - A handler
    - `declare continue handler for not found set done = true;`
    - handles exceptions/conditions, here "`not found`",
      executes a statement, here "`set done = true`", (*)
      and either "`continue`" or "`exit`" the current program

(*) 0 is the same as false. 1 is the same as true

External Language Functions/Procedures

- SQL:1999 permits the use of functions and procedures written in other languages such as C or C++
- Declaring external language procedures and functions

```
create procedure dept_count_proc(in dept_name varchar(20),
                                 out count integer)
language C
external name ' /usr/avi/bin/dept_count_proc'

create function dept_count(dept_name varchar(20))
returns integer
language C
external name '/usr/avi/bin/dept_count'
```

External Language Routines (Cont.)

- Benefits of external language functions/procedures:
  - more efficient for many operations, and more expressive power.

- Drawbacks
  - Code to implement function may need to be loaded into database system and executed in the database system's address space.
    - risk of accidental corruption of database structures
    - security risk, allowing users access to unauthorized data

Why use Stored functions and procedures?

- Stored functions and procedures (routines) can be particularly useful
  - When multiple client applications are written in different languages or work on different platforms, but need to perform the same database operations.
  - When security is paramount. Banks, for example, use stored procedures and functions for all common operations
  - In addition, you can store libraries of functions and procedures in the database server
  - Provide improved performance. Less information needs to be sent between the server and the client.
  - Tradeoff: increase the load on the database server

Triggers

Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.

- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.

Trigger Example – Referential constraint

- E.g. *time_slot_id* is not a primary key of *timeslot,* so we cannot create a foreign key constraint from *section* to *timeslot.*
- Alternative: use triggers on *section* and *timeslot* to enforce integrity constraints
- Figure 5.8 in DSC book:

```
create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
            select time_slot_id
            from time_slot)) /* time_slot_id not present in time_slot */
begin
    rollback
end;
```

Will not work in MySQL

Trigger Example – Referential constraint

- Figure 5.8 in DSC book does NOT work in MySQL because
  - Rollback is not allowed in a trigger
- The following is an alternative
  - The result is the same: an update with a time_slot_id not present in the time_slot table will not be allowed (and will thus be ignored)

```
drop trigger if exists timecheck;
delimiter //
create trigger timecheck before insert on section
for each row
begin
  if (new.time_slot_id not in (select time_slot_id from time_slot)) then
    SIGNAL sqlstate '45000' set message_text = "No no, wrong time_slot";
  end if;
end;//
delimiter ;
```

```
mysql> insert into section values
('BIO-301', '2', 'Winter', '2009', 'Painter', '514', 'I');
ERROR 1644 (45000): No no, wrong time_slot
mysql>
```

Trigger Example – Referential constraint (Cont.)

- Notice SQL and MySQL details
  - The example is a **before** rather than an **after** trigger
  - **if** (inside the block) replaces **when** (outside)
  - "**referencing new row as** *nrow*" won't work, but you can reference the new value simply with **new**
    - **new** can be used in **insert** and **update**-triggers
    - **old** can be used similarly in **delete** and **update**-triggers
  - SIGNAL is used here to "return" an error with a message
    - sqlstate '45000' means "unhandled user-defined exception."

```
drop trigger if exists timecheck;
delimiter //
create trigger timecheck before insert on section
for each row
begin
  if (new.time_slot_id not in (select time_slot_id from time_slot)) then
    SIGNAL sqlstate '45000' set message_text = "No no, wrong time_slot";
  end if;
end;//
delimiter ;
```

```
mysql> insert into section values
('BIO-301', '2', 'Winter', '2009', 'Painter', '514', 'I');
ERROR 1644 (45000): No no, wrong time_slot
mysql>
```

Trigger Example – Ad hoc constraint

- Company policy (insert on instructor trigger)
  - No new employments in high budget departments (>=90000)
  - New employees (instructors) must never have a salary greater than everybody else

```
delimiter //
create trigger instructorcheck before insert on instructor
for each row
begin
  declare s varchar(50);
  if (new.dept_name not in (select dept_name from department where budget <90000)) th
    set s=concat("No no no, no new employees in ",new.dept_name," department");
    SIGNAL sqlstate '45000' set message_text = s;
  end if;
  if (new.salary> (select max(salary) from instructor)) then
    SIGNAL sqlstate '45000' set message_text = "No no no, salary too high";
  end if;
end;//
delimiter ;
```

```
mysql> insert into instructor values (12345, 'Wong', 'Finance', 80000);
ERROR 1644 (45000): No no no, no new employees in Finance department
```

```
mysql> insert into instructor values (23456, 'Wang', 'History', 100000);
ERROR 1644 (45000): No no no, salary too high
```

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**

- Triggers can be activated before an event, which can serve as extra constraints. E.g. convert blank grades to null.

```
drop trigger if exists setnull_trigger;
delimiter //
create trigger setnull_trigger before update on takes
for each row
begin
        if (new.grade = ' ') then
                set new.grade = null;
        end if;
end; //
delimiter ;
```

Trigger to Maintain credits_earned value

- **Figure 5.9 from the DSC book**
  - **create trigger** *credits_earned* **after update of** *takes* **on** (*grade*)
    referencing new row as *nrow*
    referencing old row as *orow*
    for each row
    when *nrow.grade* <> 'F' **and** *nrow.grade* **is not null**
      and (*orow.grade* = 'F' **or** *orow.grade* **is null**)
    begin atomic
      update *student*
      set *tot_cred*= *tot_cred* +
        (**select** *credits*
        from *course*
        where *course.course_id*= *nrow.course_id*)
      where *student.id* = *nrow.id*;
    end;

Will not work in MySQL

Trigger to Maintain credits_earned value (Cont.)

- **Figure 5.9 from the DSC book:**
  - **New version that works in MySQL**

```
drop trigger if exists credits_earned;
delimiter //
create trigger credits_earned after update on takes
for each row
begin
if (new.grade <> 'F' and new.grade is not null
    and (old.grade = 'F' or old.grade is null)) then
        update student
        set tot_cred= tot_cred +
            (select credits
             from course
             where course.course_id= new.course_id)
        where student.id = new.id;
        end if;
end;//
delimiter ;
```

  - **NOTICE: "update of *takes* on (*grade*)"** is not supported in MySQL
  - **But we can simply use "`update on takes`"**

Trigger example

- **An update** trigger ensuring that amount on account always satisfies 0 ≤ amount ≤ 100

```
delimiter //
create trigger upd_check before update on account
for each row
begin
   if new.amount < 0 then
      set new.amount = 0;
   elseif new.amount > 100 then
      set new.amount = 100;
   end if;
end;//
delimiter ;
```

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction.
  - Can be more efficient when dealing with SQL statements that update a large number of rows
- Supported by some DBMS' using
  - **for each statement** instead of **for each row**

- **Insertion of 887000 rows:**

```
insert into movie.movie
  select id, title, production_year
  from imdb_movie.movie
  where kind_id=1;
```

- **with row-level: 887000 actions, with statement level: 1 action**

- **Statement Level not supported in MySQL**

A problem and a solution

- No table or array data type for output from functions and procedures
- However, you can use the following simple approach to provide relational data output from stored procedures

SQL Procedures

- A ***keyword_title*** search procedure for **`imdb_movie`**
- looking up the first title of a movie with a specific assigned a given keyword
- MySQL variable: `@title`

```
use troels;
drop procedure if exists keyword_title;
delimiter //
create procedure keyword_title(in w varchar(100), out t varchar(100))
begin
  select title into t
  from imdb_movie.movie, imdb_movie.movie_keyword, imdb_movie.keyword
  where (movie.id,keyword.id)=(movie_id,keyword_id) and keyword=w limit 1;
end //
delimiter ;
```

```
CALL keyword_title('elephant',@title);
select @title;

+ ----------- +
| @title      |
+ ----------- +
| South Africa |
+ ----------- +
1 rows
```

SQL Procedures

- So what if you want a find procedure for finding all titles?

- Problem
  - out-parameters can only be single values NOT tables

- Solution
  - A procedure with a SELECT, but no INTO, will return a result set when called
  - so simply remove out-parameter and INTO

```
use troels;
drop procedure if exists keyword_title;
delimiter //
create procedure keyword_title(in w varchar(100), out t varchar(100))
begin
  select title into t
  from imdb2016.title, imdb2016.movie_keyword, imdb2016.keyword
  where (title.id,keyword.id)=(movie_id,keyword_id) and keyword=w limit 1;
end //
delimiter ;
```

SQL Procedure, with result set

- A procedure with a SELECT, but no INTO, will return a result set

```
use troels;
drop procedure if exists keyword_title;
delimiter //
create procedure keyword_title(in w varchar(100))
begin
  select title
  from imdb2016.title, imdb2016.movie_keyword, imdb2016.keyword
  where (title.id,keyword.id)=(movie_id,keyword_id) and keyword=w;
end //
delimiter ;
```

```
> CALL keyword_title('elephant')

+ ---------- +
| title      |
+ ---------- +
| South Africa |
| Financial Weapons of Mass Destruction/T Boone Pickens/Gor
| The Bollo Caper |
| Charm and Charities |
| Snake in the Grass |
| Intia      |
| All Tied Up/Tennis Court |
| Fishing Trip/Mr. Factory |
```

SQL Procedure, with result set

- A procedure with a SELECT, but no INTO, will return a result set
- Will also work when called from a program, e.g. C# and ADO.NET or Java and JDBC

SQL Procedure, with result set

- Will also work when called from a program, here with C#, ADO.NET

```
using MySql.Data.MySqlClient;
using System;

namespace ADOExample
{
   class Program
   {
      static void Main(string[] args)
      {
         using (var conn = new MySqlConnection(
            "server=wt-220.ruc.dk;database=university;uid=troels;pwd=xxxx;SslMode=none"))
         {
            conn.Open();
            var cmd = new MySqlCommand("CALL troels.keyword_title('elephant')", conn);
            var rdr = cmd.ExecuteReader();

            while(rdr.Read())
            {
               Console.WriteLine($"{rdr.GetString(0)}");
            }
            rdr.Close();
            conn.Close();
         }
      }
   }
}
```

Compare with s7

```
South Africa
Financial Weapons of Mass Dest
The Bollo Caper
Charm and Charities
Snake in the Grass
Intia
All Tied Up/Tennis Court
Fishing Trip/Mr. Factory
As Time Goes By
(2015-03-04)
Babar and the Adventures of Ba
```