DATABASES

Lab compendium

Table of Contents

General Information

For how to get started (setting the MySQL environment, FAQs, etc.), please check the course website.

For all labs you have to hand in a report.

Please print your lab report on paper and check that it is easy to read and understand, i.e. add explanations when needed. Do *not* e-mail reports unless explicitly told so by your lab assistant.

Refer to the course web page for further information and documentation needed for the labs.

The Jonson Brothers' database

Some lab exercises are based on a database that is used for the business of the Jonson Brothers. This section describes this database.
The Jonson Brothers is a retail company with department stores in many major US cities. The company has a large number of employees and sells a varied line of products.
The company consists of a number of stores that contain a number of departments. The company has employees, who (among other things) sell items at the different stores. Sales are registered in the sale and debit tables. Items are bought from various suppliers, who also supply parts for the company's computer equipment. Deliveries of computer parts are registered in the supply table.

Below we provide

- the contents of each table in the database,
- the ER diagram of the database.

The content of the tables

All table names are prefixed with JB **so as to not enter into conflict with tables created for other courses in your MySQL account.**

JBEMPLOYEE
An employee is identified by Id and described by Name, Salary, Birthyear and Startyear. The Id of the Manager of each employee is also supplied.

ID	NAME	SALARY	MANAGER	BIRTHYEAR	STARTYEAR
157	Jones, Tim	12000	199	1940	1960
1110	Smith, Paul	6000	33	1952	1973
35	Evans, Michael	5000	32	1952	1974
129	Thomas, Tom	10000	199	1941	1962
13	Edwards, Peter	9000	199	1928	1958
215	Collins, Joanne	7000	10	1950	1971
55	James, Mary	12000	199	1920	1969
26	Thompson, Bob	13000	199	1930	1970
98	Williams, Judy	9000	199	1935	1969
32	Smythe, Carol	9050	199	1929	1967
33	Hayes, Evelyn	10100	199	1931	1963
199	Bullock, J.D.	27000	NULL	1920	1920
4901	Bailey, Chas M.	8377	32	1956	1975
843	Schmidt, Herman	11204	26	1936	1956
2398	Wallace, Maggie J.	7880	26	1940	1959
1639	Choy, Wanda	11160	55	1947	1970
5119	Bono, Sonny	13621	55	1939	1963
37	Raveen, Lemont	11985	26	1950	1974
5219	Schwarz, Jason B.	13374	33	1944	1959
1523	Zugnoni, Arthur A.	19868	129	1928	1949
430	Brunet, Paul C.	17674	129	1938	1959
994	Iwano, Masahiro	15641	129	1944	1970
1330	Onstad, Richard	8779	13	1952	1971
10	Ross, Stanley	15908	199	1927	1945
11	Ross, Stuart	12067	NULL	1931	1932

JBDEPT
A department is identified by Id and part of a Store, described by Name, Floor and contains the employee id of the Manager of the department.

ID	NAME	STORE	FLOOR	MANAGER
35	Book	5	1	55
10	Candy	5	1	13
19	Furniture	7	4	26
20	MajorAppliances	7	4	26
14	Jewelry	8	1	33
43	Children's	8	2	32
65	Junior's	7	3	37
58	Men's	7	2	129
60	Sportswear	5	1	10
99	Giftwrap	5	1	98
1	Bargain	5	0	37
26	Linens	7	3	157
63	Women's	7	3	32
49	Toys	8	2	35
70	Women's	5	1	10
73	Children's	5	1	10
34	Stationary	5	1	33
47	JuniorMiss	7	2	129
28	Women's	8	2	32

JBSTORE
A store is identified by Id and described by City.

ID	CITY
5	941
7	946
8	945
9	941

JBCITY
A city is identified by its Name and described by its State.

ID	NAME	STATE
900	Los Angeles	Calif
946	Oakland	Calif
945	El Cerrito	Calif
303	Atlanta	Ga
941	San Francisco	Calif
021	Boston	Mass
752	Dallas	Tex
802	Denver	Colo
106	White Plains	Neb
010	Amherst	Mass
981	Seattle	Wash
609	Paxton	Ill
100	New York	NY
921	San Diego	Calif
118	Hickville	Okla
841	Salt Lake City	Utah
537	Madison	Wisc

JBITEM
An item is identified by Id and described by Name, the Dept where it is sold, the Price, quantity on hand (QOH) and the identifier of the Supplier that supplied it.

<u>ID</u>	NAME	DEPT	PRICE	QOH	SUPPLIER
26	Earrings	14	1000	20	199
118	Towels, Bath	26	250	1000	213
43	Maze	49	325	200	89
106	Clock Book	49	198	150	125
23	1 lb Box	10	215	100	42
52	Jacket	60	3295	300	15
165	Jean	65	825	500	33
258	Shirt	58	650	1200	33
120	Twin Sheet	26	800	750	213
301	Boy's Jean Suit	43	1250	500	33
121	Queen Sheet	26	1375	600	213
101	Slacks	63	1600	325	15
115	Gold Ring	14	4995	10	199
25	2 lb Box, Mix	10	450	75	42
119	Squeeze Ball	49	250	400	89
11	Wash Cloth	1	75	575	213
19	Bellbottoms	43	450	600	33
21	ABC Blocks	1	198	405	125
107	The `Feel' Book	35	225	225	89
127	Ski Jumpsuit	65	4350	125	15

JBSALE
A sale can contain a number of Items, the transaction identifier Debit and the Quantity for each sold item on this receipt. For example: Debit transaction 100581 consists of two items: Item 118 with a quantity of 5 and Item 120 with quantity 1.

<u>DEBIT</u>	<u>ITEM</u>	QUANTITY
100581	118	5
100581	120	1
100582	26	1
100586	127	3
100586	106	2
100592	258	1
100593	23	2
100594	52	1

JBDEBIT
A debit (receipt of a sale) is identified by its Id and described by the timestamp Sdate when the debit took place, the Employee who sold the item, and a customer Account to which the amount was debited.

<u>ID</u>	SDATE	EMPLOYEE	ACCOUNT
100581	15-JAN-95 12:06:03	157	10000000
100582	15-JAN-95 17:34:27	1110	14356540
100586	16-JAN-95 13:53:55	35	14096831
100592	17-JAN-95 09:35:23	129	10000000
100593	18-JAN-95 12:34:56	35	11652133
100594	19-JAN-95 10:10:10	215	12591815

JBSUPPLIER
A supplier (of items and parts) is identified by Id and described by Name and City.

ID	NAME	CITY
199	Koret	900
213	Cannon	303
33	Levi-Strauss	941
89	Fisher-Price	021
125	Playskool	752
42	Whitman's	802
15	White Stag	106
475	DEC	010
122	White Paper	981
440	Spooley	609
241	IBM	100
62	Data General	303
5	Amdahl	921
20	Wormley	118
67	Edger	841
999	A E Neumann	537

JBPARTS
A part, used internally by the store, not sold to customers, is identified by Id and described by Name, Color, Weight, and quantity on hand (QOH).

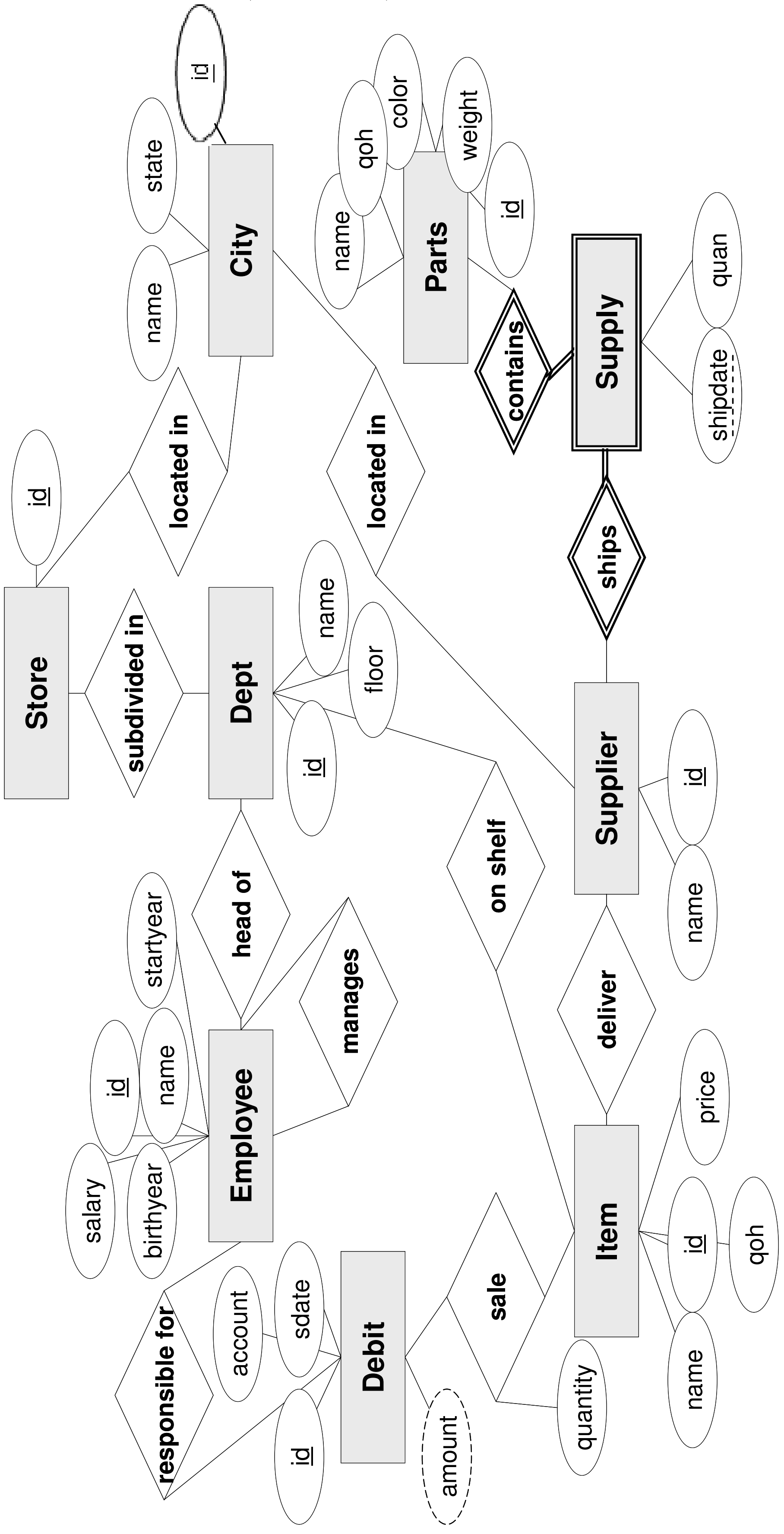
ID	NAME	COLOR	WEIGHT	QOH
1	central processor	pink	10	1
2	memory	gray	20	32
3	disk drive	black	685	2
4	tape drive	black	450	4
5	tapes	gray	1	250
6	line printer	yellow	578	3
7	l-p paper	white	15	95
8	terminals	blue	19	15
13	paper tape reader	black	107	0
14	paper tape punch	black	147	0
9	terminal paper	white	2	350
10	byte-soap	clear	0	143
11	card reader	gray	327	0
12	card punch	gray	427	0

JBSUPPLY
A Supplier supplies Part on Shipdate with quantity Quan.

SUPPLIER	PART	SHIPDATE	QUAN
475	1	1993-12-31	1
475	2	1994-05-31	32
475	3	1993-12-31	2
475	4	1994-05-31	1
122	7	1995-02-01	144
122	7	1995-02-02	48
122	9	1995-02-01	144
440	6	1994-10-10	2
241	4	1993-12-31	1
62	3	1994-06-18	3
475	2	1993-12-31	32
475	1	1994-07-01	1
5	4	1994-11-15	3
5	4	1995-01-22	6
20	5	1995-01-10	20
20	5	1995-01-11	75
241	1	1995-06-01	1
241	2	1995-06-01	32
241	3	1995-06-01	1
67	4	1995-07-01	1
999	10	1996-01-01	144
241	8	1995-07-01	1
241	9	1995-07-01	144
89	3	1995-07-04	1000
89	4	1995-07-04	1000

The Jonson Brothers' ER-Diagram

Remember to add cardinalities (1:N, N:M, etc.).



Lab 1: MySQL

Objectives

The purpose of this exercise is to practise writing queries in MySQL, including the use of aggregate functions and views.

Background Reading

Check the lecture material and study the MySQL chapter(s) in your database book.

Setting up your environment and database

To set up your computer and load the database on which you will work for the three coming labs, follow the instructions from the page *Labs—Settings* on the web page of the course.

The Lab

Formulate in MySQL.

1) List all employees, i.e. all tuples in the JBEMPLOYEE relation.
2) List the name of all departments in alphabetical order. Note: by “name” we mean the NAME attribute for all tuples in the JBDEPT relation.
3) What parts are not in store, i.e. QOH = 0? (QOH = Quantity On Hand)
4) Which employees have a salary between 9000 and 10000?
5) What was the age of each employee when they started working (STARTYEAR)?
6) Which employees have a last name ending with “son”?
7) Which items have been delivered by a supplier called *Fisher-Price*? Formulate this query using a subquery in the where-clause.
8) Formulate the same query as above, but without a subquery.
9) What is the name and color of the parts that are heavier than a card reader? Formulate this query using a subquery in the where-clause. (The MySQL query must not contain the weight as a constant.)
10) Formulate the same query as above, but without a subquery. (The query must not contain the weight as a constant.)
11) What is the average weight of black parts?
12) What is the total weight of all parts that each supplier in Massachusetts (“Mass”) has delivered? Retrieve the name and the total weight for each of these suppliers. Do not forget to take the quantity of delivered parts into account.
13) Create a new relation (a table) that contains the items that cost less than the average price for items! Define primary and foreign keys in your table!
14) Create a view that contains the items that cost less than the average price for items! *What is the difference between (13) and (14)?*

15) Create a view that calculates the total cost of each sale, by considering price and quantity of each bought item. (To be used for charging customer accounts). Should return the sale identifier (debit) and total cost.

16) Oh no! An earthquake!

   a) Remove all suppliers in Los Angeles from the table JBSUPPLIER. *What happens when you try to do this? Why? Explain the specific error message.*

   b) Which tuples make the delete-statement fail? List them using MySQL.

17) An employee has tried to find out which suppliers have delivered items that have been sold. He has created a view and a query that shows the number of items sold from a supplier.

```
mysql> create view jbsale_supply(supplier, item, quantity) as
    -> select jbsupplier.name, jbitem.name, jbsale.quantity
    -> from jbsupplier, jbitem, jbsale
    -> where jbsupplier.id = jbitem.supplier
    -> and jbsale.item = jbitem.id;
Query OK, 0 rows affected (0.01 sec)

mysql> select supplier, sum(quantity) sum from jbsale_supply
    -> group by supplier;
+--------------+------+
| supplier     | sum  |
+--------------+------+
| Cannon       |    6 |
| Koret        |    1 |
| Levi-Strauss |    1 |
| Playskool    |    2 |
| White Stag   |    4 |
| Whitman's    |    2 |
+--------------+------+
6 rows in set (0.00 sec)
```

The employee would also like to find suppliers that have *not* had any of their delivered items sold. Help him! Drop and redefine `jbsale_supply` to consider suppliers that have delivered items that have never been sold as well.
Hint: The above definition of `jbsale_supply` uses an (implicit) inner join that removes suppliers that have not had any of their delivered items sold.

Handing in

- For each problem above, list the query followed by the result of the query.
- Written answers for questions *14)* and *16a).*

Lab 2: ER and Relational Modelling

Objectives

The purpose of this exercise is to give a good understanding of database design and entity-relationship modelling.

Background Reading

Read lecture material on ER- and EER-modelling, on the translation of EER-diagrams into relational tables, and MySQL for creating tables and managing constraints.

The Lab

The Jonson Brothers' business is expanding and the database is continuously being extended with new information. The current state of the company database can be seen in the ER-diagram provided on page 8. The management of Jonson Brothers' has hired you to help them extend their database. The work requires extensions to support a bonus system where managers can be given an extra bonus (e.g. if their department have met their sale predictions) added to their salary. The management also wants to encourage customers to shop more by creating a credit card that users can use when paying for items that they buy.

1) Analyze the ER-diagram on page 8 (available electronically from the course web page) and the relational database and add information about the cardinality of the relationships such as one-to-one, one-to-many, many-to-one, and many-to-many to the ER-diagram.

2) Extend the ER-diagram with an entity type *manager* that is a subclass of *employee*. Add support for a manager bonus that is added to the salary by giving the manager entity a bonus attribute. Use the manager-entity (instead of employee) in appropriate, existing relationships. Draw your extensions to the ER-diagram and translate them to the relational model.

3) Implement your extensions in the database by first creating tables, if any, then populating them with manager data, then adding/modifying foreign key constraints. Note that some managers are managers of departments, some managers are managers of other employees, and some are both.

   We recommend to always create InnoDB tables. To create a InnoDB table, you just have to add "engine=InnoDB" after the usual create table statement. That is, CREATE TABLE customers (a INT, b CHAR (20)) ENGINE=InnoDB; If you do not explicitly create your tables as InnoDB, they will be created as MyISAM by default and you may not be able to create foreign keys. To read more about the different storage engines in MySQL, see chapter 13 in the MySQL manual.

4) All departments showed good sales figures last year! Give all current department managers $10,000 in bonus. This bonus is an addition to possible other bonuses they have.
   Hint: Not all managers are department managers. Update all managers that are referred in the JBDEPT relation.
   Note: If you have disabled MySQL's auto commit feature by issuing

set autocommit = 0 you can use **commit;** to save your changes when you are satisfied, and **rollback;** to undo your changes while you are not.

5) In the existing database, customers can buy items and pay for them, as reflected by the sale and debit tables. Now, you want to create support for storing customer information. The customers will have accounts, where they can deposit and withdraw money. The requirements are:

   Customers are stored with name, street address, city, and state. Use existing city information!

   A customer can have several accounts.

   Accounts are stored with account number, balance, and allowed credit. No customer account must have a credit above $10,000.

   Information about transactions (withdrawals/deposits/sales) such as transaction number, account number, date and time of deposit/withdrawal/sale, amount, and the employee responsible for the transaction (that is, the employee that registers the transaction, not the customer that owns the account). Thus, transaction effectively replaces/extends the existing debit-entity.

   a) Extend the EER-diagram with your new entities, relationships, and attributes. Translate the extensions to the diagram into the relational model.

   b) Implement your extensions in your database. Add primary key constraints, foreign key constraints and integrity constraints to your table definitions. Do not forget to correctly set up the new and existing foreign keys.

- Use `alter table t1 drop foreign key constraint_name;`
  and
  ```
  alter table t2
  add constraint constraint_name
  foreign key (t2_attribute) references t1(t1_attribute);
  ```
  to change existing foreign keys.

Handing in

- ER diagram with *cardinalities* and the *extensions(to both the ER and the relational models)* required by questions *2)* and *5.a)*
- The MySQL commands to create, modify and delete tables and constraints (such as foreign keys), as well as the results of these commands.

Lab 3: BrianAir Database

Objectives

This lab combines what you have learnt in the previous labs in one larger project.

Background Reading

For this lab, you need to be familiar with EER-modelling, translation of EER into relational tables, MySQL queries, stored procedures, triggers and transactions, that is, almost the whole course.

The Lab

The low price airline BrianAir asks you to assist them with designing their flight and booking database. In addition, you have to implement the backend for their web interface for booking flight, i.e. you provide stored procedures and MySQL queries that will be embedded in the web middleware.

1) Draw an EER-diagram for BrianAir's database.
   a. BrianAir uses only one type of airplane that takes 60 passengers. You need not model different airplane types.
   b. BrianAir currently only flies between Lillby and Smallville (and returns) but they will soon expand.
   c. BrianAir operates on a strict weekly schedule. There are no exceptions for holidays. The weekly schedule is valid for one year and may be changed on every January 1st. Of course, there can be several flights per day.

      A simple solution would be to model flights as an entity with attributes such as cities of departure and arrival, day of the year and time of departure, etc. However, **this is not acceptable** because this table may contain a lot of duplicated information. Use two additional entities instead:

      - Route, which contains all the routes the company flies. A route is characterized by the cities of departure and arrival.
      - Weekly flights, which contains the schedule of flights for any week since BrianAir flies the same flights every week of the year. A weekly flight is characterized by a route, plus a day of the week, plus the time of departure.
      - Year should also be involved somehow, as you are about to discover.

   d. The flight pricing depends on
      - the start and stop destination which has a base price,
      - the day of the week. BrianAir has the same weekday pricing factor for all flights regardless of destination, e.g. factor 4.7 on Fridays and Sundays, factor 1 on Tuesdays, etc.

- the number of already confirmed/booked passengers on the flight. The more passengers are booked the more expensive becomes the flight.

The price is thus calculated as:

$\text{baseprice}_{\text{to, from}} \cdot \text{weekdayfactor}_{\text{day}} \cdot \max(1, \text{pass\#}_{\text{flight}})/60 \cdot \text{passengerfactor}$

Pricing factors and base prices can change when the schedule changes, once per year!

e. BrianAir only issues tickets to passengers older than 18 years. Passengers can be adults (older than 18 years of age), children, and infants (younger than 2 years of age). Infants travel at no cost and do not receive a seat (sit in parent's lap).

f. Customers cannot book more than one year in advance.

g. It is possible to book several people in one booking. One passenger of a booking is the *contact* and must supply phone number and e-mail address.

h. It is a normal case that the one who pays for the flight (the credit card holder) does not necessarily participate in the flight.

i. The payment of the flight is confirmed by issuing a unique ticket number that the passenger needs to bring to the airport instead of a paper ticket.

j. Customers must not end up in a deadlock situation, and double payments are not allowed. However, BrianAir is afraid of denial-of-service attacks to its reservation system and does not want that unpaid and pending reservations block paying customers. The airline demands a reservation strategy that fails in favour of BrianAir: It is possible to reserve seats on a flight that already has many reserved seats; but it is not possible to reserve seats on a flight where all seats are already paid for. At payment time, it must be checked that the reserved seats are really available. If they are not available, the whole booking is aborted and the customer needs to restart from scratch.

k. The actual price for a booking is calculated at payment or booking time (as discussed above, the price may be higher at payment time).

l. If you want, every person in one booking can get the same seat price.

Other requirements are up to your assumptions. State them in the EER-diagram. Also read *3)* and *4)*—they may clarify things further.

2) Translate the EER-diagram into a relational model.

Before proceeding,
hand in your EER-diagram and translations.

Reading the description of the rest of the exercises in the lab may help you to draw your EER-diagram, though.

Only projects with a previously approved EER-diagram and translations are considered when lab assistants check your final project report.

3) Implement your relational model in MySQL. We recommend to always create InnoDB tables. To create a InnoDB table, you just have to add "engine=InnoDB" after the usual create table statement. That is, CREATE TABLE customers (a INT, b CHAR (20)) ENGINE=InnoDB; If you do not explicitly create your tables as InnoDB, they will be created as MyISAM by default and you may not be able to create foreign keys. To read more about the different storage engines in MySQL, see chapter 13 in the MySQL manual.

   It is now time to fill your database with data. The goal is to arrive at a system that allows the booking of flights. Make sure that you have a simple but working system. Improve later.

   Create INSERT-statements and STORED PROCEDURES for

   a. setting up new destinations (INSERT),
   b. *For your own convenience, create a script that removes all BrianAir tables from your account and installs them again with suitable data. You need this for testing, reinstallation and in case of having to demonstrate your project. Keep this script updated as you proceed. To create a script, create a text file myscript.sql using any text editor. To run the script in MySQL, just type "source myscript.sql".*
   c. setting up a weekly flight schedule for this and next year (INSERT) and creating the actual flight details from this schedule for one year and one day in advance (STORED PROCEDURE). The idea is that you insert routes and weekly flights by hand and, then, use these to create the actual flights automatically. That is, you have to iterate through the days of the year and use weekly flights for the relevant information.

      Depending on your modelling, this may be very simple or the most difficult procedure of this lab. Do not get stuck on this exercise. You may want to fill the flight table with only a few flights first to be able to proceed. Or you may want to implement a simplified version first, e.g. only create Sunday flights. This procedure may need date functions. Check them out in the lab material at the end of the compendium.
   d. setting up a pricing policy for this and next year (INSERT),
   e. reserving seats for a given number of passengers on a specified flight taking into account the strategy described above (STORED PROCEDURE). Seat reservation results in a session number that can (and must) be referred below. Session number and ticket number are not the same! Mind this difference.
   f. adding passenger details (mr/ms, first name, surname, possibly adult/child/infant and at least one contact info per booking) to a booking (STORED PROCEDURE). Implement a check so that one cannot add more

passengers—except infants if you modelled them—to a booking than there are reserved seats in the booking created. Implement a check so that you have at least one contact per booking.

g. adding payment details such as amount, name as on credit card, credit card type, expiry month, expiry year, credit card number. *How can you protect the credit card information in the database from hackers? You need not implement this protection.* (STORED PROCEDURE),

h. confirming previously reserved seats, calculating the actual price of the booking and returning a unique ticket number (the flight is now paid for). The actual payment transaction takes place in the database of the credit card company and need not be considered here. Take into account that the seat reservation may have been invalidated due to other paid bookings and that you may have to abort the whole booking (see above for the strategy description) (STORED PROCEDURE). Implement a check that you cannot confirm a booking without payment details.

4) Given a destination, number of prospective passengers and a date, show all flights for this date with their price and the number of available seats according to the reservation strategy. The result of this query is used by customers to find available flights. You may create views to solve this query.

5) Open a new MySQL session in a new window. We call the original session A and the new session B. Do not forget to **set autocommit = 0** in both sessions.

   a. In session A, add a new booking (step e in exercise 3).

   b. Is this booking visible in session B ? Why ?

   c. What happens if you try to modify the booking from A in B ? Explain.

6) Let two customers book seats on the same flight at the same time using two different mysql sessions with **set autocommit = 0** in both sessions. Make sure that there are not enough seats available for both customers (only one customer can succeed). Use your stored procedures from *3)* and experiment with COMMIT, LOCK TABLES, ROLLBACK, SAVEPOINT, SELECT...FOR UPDATE and START TRANSACTION until the strategy of 1.*j* (no deadlocks, no double payments) is satisfied. You can find a description of these commands at the end of this compendium. ***Do NOT modify your procedures*** *but put transactional commands* ***around*** *your procedure calls. Deadlocks in MySQL usually time out within a minute.*

   You may get some help from the exercise 5 above: To avoid double payments, every customer must be able to correctly count the number of unpaid seats in the corresponding flight. How can you ensure this if autocommit is 0 and, thus, the customers may have not committed the payments they have just done, e.g. a customer may have paid but not committed yet because he/she is taking a break, so the other customers cannot see his/her payment ? Solution: Use the commands mentioned above. Yet another way to see the problem is that several customers trying to book simultaneously will run the same sequence of MySQL instructions. Is there any way that the instructions run by different customers get interleaved so that a double payment occurs ? How would you avoid this harmful interleaving ? Of course, you want to allow that instructions interleave as much as possible to allow customers booking simultaneously. However, certain parts of the sequence of instructions may have to occur atomically to avoid double payments. Use the commands above to guarantee this.

Handing in

Step 1: At the beginning of the project

- EER-diagram,
- Relational model.

Step 2: At the end of the project.

- ***Approved*** **EER-diagram,**
- MySQL-code for creating the tables and constraints on them,
- Code for all stored procedures, triggers, functions etc.
- *Example runs* for
    - filling the database with flights,
    - booking a flight.
- Code for *4)*.
- Written answers to *5)*.
- Demonstrate *6)* by showing how the two transactions interleave.
- Show/discuss how you have addressed all the points in the description above.
- Finally, send an email with the MySQL code as an attachment to jospe.liu@analys.urkund.se. The name of the attached file should be as student1student2.txt, where student1 and student2 are the login names of the students in the group. The electronic version will then be run through a system that compares the code with solutions from other students and from previous years. The teachers get an overview of this analysis. In this course we would like to keep the possibility of having a project where the students work freely and very much on their own. Therefore we use this method for analysis of project reports instead of putting higher restrictions on when and where you have to work with your projects. Sending the referred email is mandatory for achieving your grade in the lab course.

If the lab assistant does not understand your report, you will be contacted for a question and answer session and a possible demonstration of your project.
Therefore, do not delete any database objects before your lab is approved in the lab system!

Lab 4: Graphical Interface to Lab 3

In lab 3, you implemented the back-end of the BrianAir database. In this lab, you are requested to implement the front-end or graphical interface to the database. You can use any programming language, though your lab assistant may not be able to help you with it. We recommend you to use PHP, which is available at IDA. For more information on how to get started with PHP, please visit the course website.

Hand in the code for your interface, plus some screen-shots that show how to operate it.

Lab 5: Connecting Several Lab 3

Two groups of students, say A and B, will connect their implementations of the BrianAir database to simulate that their two airlines have gone into partnership. This will allow passengers to fly to more destinations by connecting flights from one airline with those of the other airline. For instance, one could first fly with BrianAir A from City1 to City2 and then from City2 to City3 with BrianAir B. Specifically,

- You have to allow passengers to buy tickets for the flights run by both companies.
- You have to allow passengers to reach a destination if there is a sequence of flights that can take them there. Of course, some of the flights in the sequence may be operated by BrianAir A and some by BrianAir B. You have to guarantee that the connections are feasible.

For carrying out this lab, you have to connect the databases A and B by granting rights from one to another (see section 12.5.1.3 of the MySQL manual or the appendix at the end of this compendium for the syntax of the GRANT command). You also need to implement new routines that allow flights with connections to be booked.

It is not allowed to implement both databases A and B in the same MySQL account. It is not allowed to change the ER diagrams of the databases A and B. If you believe you really need to do so to carry out this lab, please contact first your lab assistant.

If you want to implement this lab but have not found another group to collaborate with, then you can create two instances of your BrianAir database, one in the MySQL account of each of the two members of the group, and connect them as described above.

Hand in the new MySQL code at the end of the lab.

Simplified MySQL Syntax

This section contains simplified syntax descriptions for MySQL. Check the online documentation for the complete syntax. The following commands are presented here:

ALTER TABLE
COMMIT
CREATE FUNCTION
CREATE INDEX
CREATE PROCEDURE
CREATE TABLE
CREATE TRIGGER
CREATE VIEW
DELETE
DROP
GRANT
INSERT
LOCK TABLES
RENAME
REVOKE
ROLLBACK
SAVEPOINT
SELECT
START TRANSACTION
UPDATE

Formatting conventions:

BOLD ARIAL	language element, example: **COMMIT**.
Regular font	a name that you will have to provide
Italics	a placeholder for a syntax that is shown elsewhere.

ALTER TABLE

Redefines the columns or constraints of an existing table.

ALTER TABLE tbl_name *alter_specification* [, *alter_specification*] ...;

alter_specification:

ENGINE = {**InnoDB**|**MyISAM**}
| **ADD** [**COLUMN**] *column_definition* [**FIRST** | **AFTER** col_name]
| **ADD** [**COLUMN**] (*column_definition*,...)
| **ADD** {**INDEX**|**KEY**} [index_name] (*index_col_name*,...)
| **ADD** [**CONSTRAINT** [constraint_name]] **PRIMARY KEY** (*index_col_name*,...)
| **ADD** [**CONSTRAINT** [constraint_name]] **UNIQUE** [**INDEX**|**KEY**] [index_name] (index_col_name,...)
| **ADD** [**CONSTRAINT** [constraint_name]] **FOREIGN KEY** [index_name] (index_col_name,...)
[*reference_definition*]
| **ALTER** [**COLUMN**] col_name {**SET DEFAULT literal** | **DROP DEFAULT**}
| **CHANGE** [**COLUMN**] old_col_name *column_definition* [**FIRST**|**AFTER** col_name]
| **MODIFY** [**COLUMN**] old_col_name *column_definition* [**FIRST** | **AFTER** col_name]
| **DROP** [**COLUMN**] col_name
| **DROP PRIMARY KEY**
| **DROP** {**INDEX**|**KEY**} index_name
| **DROP FOREIGN KEY** constraint_name
| **RENAME** [**TO**] new_tbl_name

column_definition:

col_name *data_type* [**NOT NULL** | **NULL**] [**DEFAULT** default_value]
[**AUTO_INCREMENT**] [**UNIQUE** [**KEY**] | [**PRIMARY**] **KEY**]
[**COMMENT** 'string'] [*reference_definition*]

data_type:

BIT[(length)] | **TINYINT**[(length)] | **SMALLINT**[(length)] | **MEDIUMINT**[(length)] | **INT**[(length)] |
INTEGER[(length)] | **BIGINT**[(length)] | **REAL**[(length,decimals)] | **DOUBLE**[(length,decimals)] |
FLOAT[(length,decimals)] | **DECIMAL**(length,decimals) | **NUMERIC**(length,decimals)
| **DATE** | **TIME** | **TIMESTAMP** | **DATETIME** | **YEAR**
| **CHAR**(length) | **VARCHAR**(length)

index_col_name:

col_name [(length)] [**ASC** | **DESC**]

reference_definition:

REFERENCES tbl_name [(*index_col_name*,...)]
[**ON DELETE** *reference_option*] [**ON UPDATE** *reference_option*]

reference_option:

RESTRICT | **CASCADE** | **SET NULL** | **NO ACTION**

COMMIT

Saves the changes in the current transaction to the database. Also erases the transaction's savepoints and releases the transaction's locks.

COMMIT;

!

By default, MySQL runs with autocommit mode enabled. This means that as soon as you execute a statement that updates (modifies) a table, MySQL stores the update on disk. If you are using a transaction-safe storage engine (such as InnoDB, BDB, or NDB Cluster), you can disable autocommit mode with the following statement:

SET AUTOCOMMIT=0;

See also **START TRANSACTION.**

CREATE FUNCTION

Creates a stored function.

DELIMITER //
CREATE FUNCTION function_name (argname *data_type*)
RETURN *data_type*
BEGIN
    -- SQL/PSM code goes here;
END //
DELIMITER ;
    -- *OBS*: You may need add a space after semicolon;

Test the function by using it in an expression:

```
mysql> select myfunction(1, 'Name');
```

CREATE INDEX

Creates a new index on the specified columns in a table or cluster.

CREATE [UNIQUE] INDEX index_name [*index_type*] **ON** tbl_name (*index_col_name*,...);

index_col_name:
    col_name [(length)] [**ASC | DESC**]

index_type:
    USING {**BTREE | HASH**}

CREATE PROCEDURE

Creates a stored procedure.

DELIMITER //
CREATE PROCEDURE proc_name ([**IN|OUT|INOUT**] argname *datatype*)
RETURN *datatype*
BEGIN
    -- SQL/PSM code goes here;
END //
DELIMITER ;

datatype here is for example **INT, CHAR(20).**

Test the procedure by calling it:

```
mysql> call myproc;
```

or

```
mysql> call myproc(1, 'name');
```

CREATE TABLE

CREATE TABLE tbl_name (*create_definition*,...) [*table_option* ...]
or
CREATE TABLE tbl_name [(*create_definition*,...)] [*table_option* ...] *select_statement*

create_definition:

    column_definition
    | [**CONSTRAINT** [constraint_name]] **PRIMARY KEY** [*index_type*] (*index_col_name*,...)
    | {**INDEX**|**KEY**} [index_name] [*index_type*] (*index_col_name*,...)
    | [**CONSTRAINT** [constraint_name]] **UNIQUE** [**INDEX**|**KEY**]
        [index_name] [*index_type*] (*index_col_name*,...)
    | [**CONSTRAINT** [constraint_name]] **FOREIGN KEY**
        [*index_name*] (*index_col_name*,...) [*reference_definition*]
    | **CHECK** (expr)

column_definition:

    col_name *data_type* [**NOT NULL** | **NULL**] [**DEFAULT** default_value]
    [**AUTO_INCREMENT**] [**UNIQUE** [**KEY**] | [**PRIMARY**] **KEY**]
    [**COMMENT** 'string'] [*reference_definition*]

data_type:

    BIT[(length)] | **TINYINT**[(length)] | **SMALLINT**[(length)] | **MEDIUMINT**[(length)] | **INT**[(length)] |
    INTEGER[(length)] | **BIGINT**[(length)] | **REAL**[(length,decimals)] | **DOUBLE**[(length,decimals)] |
    FLOAT[(length,decimals)] | **DECIMAL**(length,decimals) | **NUMERIC**(length,decimals)
    | **DATE** | **TIME** | **TIMESTAMP** | **DATETIME** | **YEAR**
    | **CHAR**(length) | **VARCHAR**(length)

index_col_name:

    col_name [(length)] [**ASC** | **DESC**]

index_type:

    USING {**BTREE** | **HASH**}

reference_definition:

    REFERENCES tbl_name [(*index_col_name*,...)]
        [**ON DELETE** *reference_option*] [**ON UPDATE** *reference_option*]

reference_option:

    RESTRICT | **CASCADE** | **SET NULL** | **NO ACTION**

table_option:

    ENGINE [=] {**MyISAM** | **InnoDB**}
    | **AUTO_INCREMENT** [=] value
    | [**DEFAULT**] **CHARACTER SET** charset_name
    | **COMMENT** [=] 'string'

select_statement:

    [**IGNORE** | **REPLACE**] [**AS**] **SELECT** ...   (a legal SELECT statement)

CREATE TRIGGER

Creates a database trigger.

DELIMITER //
CREATE TRIGGER trigger_name *trigger_time trigger_event*

    ON tbl_name **FOR EACH ROW**
    BEGIN
    -- sql/psm block using **NEW**.attr or **OLD**.attr to access
    -- the attribute attr of the table tbl_name
    END//

DELIMITER;

trigger_time:

    BEFORE | **AFTER**

trigger_event:

    INSERT | **UPDATE** | **DELETE**

CREATE VIEW

Creates a new view of one or more tables and/or other views.

```
CREATE [OR REPLACE] VIEW view_name [(column_name, …)]
        AS (a valid select statement)
        [WITH CHECK OPTION]
```

DELETE

Removes rows from a table or view that meet the where-condition. Removes all rows if no where-clause is specified.

```
DELETE FROM tbl_name
  [WHERE where_condition]
```

Multiple-table syntax:

```
DELETE tbl_name[.*] [, tbl_name[.*]] ...
  FROM table_references
  [WHERE where_condition]
```

DROP

This command removes objects and constraints from the database.

```
DROP object_type [IF EXISTS] object_name
object_type:
        FUNCTION | INDEX | PROCEDURE | TABLE | TRIGGER | VIEW
```

GRANT

Gives system privileges or roles to users and roles.

```
GRANT priv_type [(column_list)] [, priv_type [(column_list)]] ...
        ON [object_type] {tbl_name | * | *.* | db_name.*}
        TO user [IDENTIFIED BY [PASSWORD] 'pwd'], [, user [IDENTIFIED BY [PASSWORD] 'pwd']] ...
        [WITH GRANT OPTION]
object_type:
        TABLE | FUNCTION | PROCEDURE
```

The *priv_type* values that you can specify for a **table** are **SELECT**, **INSERT**, **UPDATE**, **DELETE**, **CREATE**, **DROP**, **GRANT OPTION**, **INDEX**, **ALTER**, **CREATE VIEW** and **SHOW VIEW**.

The *priv_type* values that you can specify for a **column** (that is, when you use a column_list clause) are **SELECT**, **INSERT**, and **UPDATE**.

The *priv_type* values that you can specify at the **routine level** are **ALTER ROUTINE**, **EXECUTE**, and **GRANT OPTION**. **CREATE ROUTINE** is not a routine-level privilege because you must have this privilege to create a routine in the first place.

INSERT

Adds new rows to a table or view.

```
INSERT [IGNORE]
  [INTO] tbl_name [(col_name,...)]
  VALUES ({expr | DEFAULT},...)
```

LOCK TABLES

Locks base tables (but not views) for the current thread. You cannot refer to a locked table multiple times in a single query using the same name. Use aliases instead, and obtain a separate lock for the table and each alias.

```
LOCK TABLES
  tbl_name [AS alias]
    {READ [LOCAL] | [LOW_PRIORITY] WRITE}
  [, tbl_name [AS alias]
    {READ [LOCAL] | [LOW_PRIORITY] WRITE}] ...

UNLOCK TABLES
```

When you use LOCK TABLES, you must lock all tables that you are going to use in your queries. Because LOCK TABLES will not lock views, if the operation that you are performing uses any views, you must also lock all of the base tables on which those views depend. While the locks obtained with a LOCK TABLES statement are in effect, you cannot access any tables that were not locked by the statement. Also, you cannot use a locked table multiple times in a single query. Use aliases instead, in which case you must obtain a lock for each alias separately. An example follows:

```
> mysql> lock table jbemployee write;
> Query OK, 0 rows affected (0.00 sec)
> mysql> select * from jbemployee where id in (select id from jbemployee);
> ERROR 1100 (HY000): Table 'jbemployee' was not locked with LOCK TABLES
> mysql> lock table jbemployee write, jbemployee as e write;
> Query OK, 0 rows affected (0.00 sec)
> mysql> select * from jbemployee where id in (select id from jbemployee as e);
```

Check the MySQL manual (section 12.4.5) for more information on lock tables.

RENAME

Renames a table.

```
RENAME TABLE tbl_name TO new_tbl_name [, tbl_name2 TO new_tbl_name2] ...
```

REVOKE

Revokes system privileges and roles from users and roles. Reverse of the GRANT command.

```
REVOKE priv_type [(column_list)] [, priv_type [(column_list)]] ...
  ON [object_type] {tbl_name | * | *.* | db_name.*}
  FROM user [, user] ...

REVOKE ALL PRIVILEGES, GRANT OPTION FROM user [, user] ...
object_type:
        TABLE | FUNCTION | PROCEDURE
```

The *priv_type* values that you can specify for a **table** are **SELECT**, **INSERT**, **UPDATE**, **DELETE**, **CREATE**, **DROP**, **GRANT OPTION**, **INDEX**, **ALTER**, **CREATE VIEW** and **SHOW VIEW**.

The *priv_type* values that you can specify for a **column** (that is, when you use a column_list clause) are **SELECT**, **INSERT**, and **UPDATE**.

The *priv_type* values that you can specify at the **routine level** are **ALTER ROUTINE**, **EXECUTE**, and **GRANT OPTION**. **CREATE ROUTINE** is not a routine-level privilege because you must have this privilege to create a routine in the first place.

ROLLBACK

Undoes all changes made since the savepoint. Undoes all changes in the current transaction if no savepoint is specified.

ROLLBACK [**TO SAVEPOINT** identifier]

SAVEPOINT

Identifies a savepoint in the current transaction to which changes can be rolled back.

SAVEPOINT identifier
RELEASE SAVEPOINT identifier

SELECT

Queries one or more tables or views. Returns rows and columns of data. May be used as a statement or as a subquery in another statement.
Check the syntax refinements in the online documentation.

```
SELECT [ALL | DISTINCT | DISTINCTROW ] select_expr, ...
        [FROM table_references
        [WHERE where_condition]
        [GROUP BY {col_name | expr | position} [ASC | DESC], …]
        [HAVING where_condition]
        [ORDER BY {col_name | expr | position} [ASC | DESC], ...]
        [LIMIT {[offset,] row_count | row_count OFFSET offset}]
        [PROCEDURE procedure_name(argument_list)]
        [FOR UPDATE | LOCK IN SHARE MODE]]
```

START TRANSACTION

Enforces read consistency at the transaction level. Must be given at the start of the transaction.

```
START TRANSACTION [WITH CONSISTENT SNAPSHOT] | BEGIN [WORK]
```

Related:

```
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
SET AUTOCOMMIT = {0 | 1}
```

UPDATE

Changes the values of columns in rows that meet the WHERE condition in a table or view. Changes all rows if no WHERE condition is specified.

```
UPDATE [IGNORE] tbl_name
  SET col_name1=expr1 [, col_name2=expr2 ...]
  [WHERE where_condition]
  [ORDER BY ...]
  [LIMIT row_count]
```

Multiple-table syntax:

```
UPDATE [IGNORE] table_references
  SET col_name1=expr1 [, col_name2=expr2 ...]
  [WHERE where_condition]
```

Some Useful Functions

Check the online documentation for the complete list. You can test all these functions by e.g. issuing in MySQL: SELECT *function*(argument);

NUMBER FUNCTIONS

ABS(n)	Absolute value of *n*.
CEIL(n)	Smallest integer greater than or equal to *n*.
COS(n)	Cosine of *n* in radians.
EXP(n)	*e* raised to the *n*th power.
FLOOR(n)	Largest integer equal to or less than *n*.
GREATEST(a1,...,an)	Returns the largest argument.
LEAST(a1,...,an)	Returns the smallest argument.
LN(n)	Natural logarithm of *n*, where *n* >0.
LOG(m, n)	Logarithm, base *m*, of *n*.
MOD(m, n)	Remainder of *m* divided by *n*.
POWER(m, n)	*m* raised to the *n*th power.
ROUND(n[,m])	*n* rounded to *m* decimal places; *m* defaults to 0.
SIGN(n)	if *n*<0, -1; if *n*=0, 0; if *n*>0, 1.
SIN(n)	Sine of *n* in radians.
SQRT(n)	Square root of *n*; if *n*<0, NULL.
TAN(n)	Tangent of *n* in radians.
TRUNC(n,m)	*n* truncated to *m* decimal places.

DATE FUNCTIONS

ADDDATE(d, n)	Date *d* plus *n* days.
ADDTIME(d, n)	Date and time *d* plus time expression *n*.
CONVERT_TZ(d, t1, t2)	Returns the date *d* of time zone *t1* in *t2*.
CURDATE()	current date (no time).
CURTIME()	current time (no date).
DATE_ADD(d, i)	adds an interval *i* to date *d*.
DATE_SUB(d, i)	subtracts an interval *i* from date *d*.
NOW()	current date and time at start of statement execution
SYSDATE()	absolutely current date and time

DATE FORMATTING

DATE_FORMAT(d, fmt) returns formatted output
Most important format models for DATE_FORMAT:

%a	Abbreviated weekday name (Sun..Sat)
%b	Abbreviated month name (Jan..Dec)
%c	Month, numeric (0..12)
%d	Day of the month, numeric (00..31)
%e	Day of the month, numeric (0..31)
%f	Microseconds (000000..999999)
%H	Hour (00..23)
%h	Hour (01..12)
%i	Minutes, numeric (00..59)
%j	Day of year (001..366)
%k	Hour (0..23)
%l	Hour (1..12)
%M	Month name (January..December)
%m	Month, numeric (00..12)
%p	AM or PM
%r	Time, 12-hour (hh:mm:ss followed by AM or PM)
%S	Seconds (00..59)
%T	Time, 24-hour (hh:mm:ss)
%U	Week (00..53), where Sunday is the first day of the week
%u	Week (00..53), where Monday is the first day of the week
%V	Week (01..53), where Sunday is the first day of the week; used with %X
%v	Week (01..53), where Monday is the first day of the week; used with %x

%W	Weekday name (Sunday..Saturday)
%w	Day of the week (0=Sunday..6=Saturday)
%X	Year for the week where Sunday is the first day of the week, numeric, four digits; used with %V
%x	Year for the week, where Monday is the first day of the week, numeric, four digits; used with %v
%Y	Year, numeric, four digits
%y	Year, numeric (two digits)
%%	A literal '%' character

Examples:

```
SELECT DATE_FORMAT('1997-10-04 22:23:00', '%W %M %Y'); → 'Saturday October 1997'
SELECT DATE_FORMAT('1997-10-04 22:23:00', '%H:%i:%s'); → '22:23:00'
SELECT DATE_FORMAT('1997-10-04 22:23:00','%D %y %a %d %m %b %j');
        → '4th 97 Sat 04 10 Oct 277'
SELECT DATE_FORMAT('1997-10-04 22:23:00', '%H %k %I %r %T %S %w');
        → '22 22 10 10:23:00 PM 22:23:00 00 6'
SELECT DATE_FORMAT('1999-01-01', '%X %V'); → '1998 52'
```

CHARACTER FUNCTIONS RETURNING CHARACTERS

CHAR(n,…)	Character with numeric value *n*.
CONCAT(c1, c2, …)	Concatenates *c1*, *c2* etc.
LOWER(char)	*char* with all letters in lower case.
LPAD(char1, n, char2)	*char1* left-padded to display length *n* with the sequence of characters in *char2*.
LTRIM(char)	*char* with leading blanks removed.
REPLACE(char, searchstring, replacement)	*char* with every occurrence of *searchstring* replaced by *replacement*.
RPAD(char1, n, char2)	as LPAD but for right-padding.
RTRIM(char)	as LTRIM but for removing last characters.
SUBSTRING(…)	advanced handling of substrings.
TRIM(char)	removes trailing and leading blanks in *char*.
UPPER(char)	*char* with all letters in upper case.

CHARACTER FUNCTIONS RETURNING NUMERIC VALUES

ASCII(char)	Returns a decimal number equivalent to the value of the first character of *char* in the database character set.
CHAR_LENGTH(char)	Length of *char* in characters. Use this for multi-byte characters.
INSTR(char1, char2)	Position of the occurrence of *char2* in *char1*.
LENGTH(char)	Length of *char* in bytes.

OTHER FUNCTIONS

CASE expr WHEN search1 THEN return1, [WHEN search2 THEN return2,]…[ELSE default])	If *expr* equals any *search*, returns the following *return*; if not, returns *default*.
DEFAULT(col)	returns the default value of table column *col*.
IFNULL(expr1, expr2)	returns *expr2* if *expr1* is null.
SLEEP(n)	sleep *n* seconds.
USER()	Name of the current user.