CS 683 Emerging Technologies: Embracing Change
Spring Semester, 2001
Doc 23 Integration & Some MySQL
Contents

Integration 2
MySQL 5
- Database & tables 5
- MySQL Names 6
- MySQL Data Types 8
- MySQL Columns Types 9
  - Numeric 9
  - String Column Types 10
  - Date & Time Column Types 11
- Basic SQL Commands 12
- Indexing 13
- Operators 17

Using MySQL In Squeak 22
- Using MySQL Squeak Driver 23
- Statements 27
- MySQL & Smalltalk Types 31
  - INT Types 32
  - DOUBLE & FLOAT 33
  - CHAR, VARCHAR, *TEXT 34
  - *BLOB 35
  - DATE 36
  - TIME 37
  - DATETIME 38
  - ENUM 39

References

MySQL, Paul DuBois, New Riders Publishing, 2000.
This is a very good book. A number of examples and tables in this lecture are from this text.

On-line MySQL Manual at: http://www.mysql.com/documentation/index.html

Copyright ©, All rights reserved. 2001 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.

Integration

Situation

XP team with N members working on a project

At the end of an iteration

We have a code base, call it Base1

We start a new iteration

All pairs check out Base1 and start to work

A pair finishes a task and integrates changes into code base

We how have a new code base, Base2

When the next pair finishes a task:

- They have changes to Base1
- They integrate into Base2
- They are not done integrating until all tests pass
- They have to make sure that the first pairs changes still work

In general we could have a pair that:

- Has changes to Base1
- Integrates their changes into BaseK

The larger the K the harder the integration could be

Avoid Integration Hell

Use some form of source code configuration management

Integrate often

MySQL
Database & tables

Database consists of a number of tables

Table is a collection of records

Each Column of data has a type

firstname	lastname	phone	code
John	Smith	555-9876	2000
Ben	Oker	555-1212	9500
Mary	Jones	555-3412	9900

MySQL Names

Databases, tables columns & indexes have names

Legal Characters

Alphanumeric characters
'_'
'$'

Names can start with digits

Name length

Up to 64 characters tables, databases, columns & indexes

Name qualifiers

A table is in a database

Full name of a table is databaseName.tableName

A column is in a table

Full name of a table is
databaseName.tableName.columnName

Often the full name is not needed

Example of Nonqualified Names

```
# Set a default database

USE acm;

/* Now select some columns */

SELECT last_name , first_name FROM members;
```

acm is a database
members is a table in the acm database
last_name & first_name are columns in members

Case Sensitivity

SQL keywords and function names

- Not case sensitive

Database & table names

- Are implemented using directories and files
- Case sensitivity depend on OS

Column and index names

- Not case sensitive

MySQL Data Types

- Numeric Values
  Integer - decimal or hex
  Floating-point - scientific & 12.1234

- String Values
  Use single or double quotes
  "this is a string"
  'So is this'

Sequence	Meaning
\0	NUL (ASCII 0)
\'	Single quote
\"	Double quote
\b	Backspace
\n	Newline
\r	Tab
\\	Backslash

  Including a quote character in a string
  Double quote the character
    'Don''t do it'
    "He said, ""Go home"" "

  Use the other quote character
    "Don't do it"
    'He said, "Go home" '

  Escape the quote character with a backslash

- Date and Time
- NULL

MySQL Columns Types
Numeric

Type	Range
TINYINT[(M)]	Signed Values: -128 to 127 Unsigned Values: 0 to 225
SMALLINT[(M)]	Signed Values: -32,768 to 32,767 Unsigned Values: 0 to 65,535
MEDUIMINT[(M)]	Signed Values: -8,388,608 to 8,388,607 Unsigned Values: 0 to 16,777,215
INT[(M)]	Signed Values: -2,147,683,648 to 2,147,683,647 Unsigned Values: 0 to 4,294,967,259
BIGINT[(M)]	Signed Values: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 Unsigned Values: 0 to 2^{32}-1
FLOAT[(M,D)], FLOAT(4)	MIN VALUES: ±1.175494351E-38 MAX VALUES: ±3.402823466+38
DOUBLE[(M,D)], FLOAT(8)	MIN VALUES: ±2.22507E-308 MAX VALUES: ±1.79769+308
DECIMAL(M,D)	Depends on M & D

Ints & Floats

M = number of digits to the left of the decimal displayed
D = number of decimal places displayed

M & D do not affect how the number is stored

DECIMAL

Stored as a string
M & D determine how many characters are stored

String Column Types

Type	Max Size
CHAR(M)	M (<=225) bytes
VARCHAR(M)	M (<=225) bytes
TINYBLOB, TINYTEXT	2^8-1 bytes
BLOB, TEXT	2^{16}-1 bytes
MEDIUMBLOB, MEDIUMTEXT	2^{24}-1 bytes
LONGBLOB, LONGTEXT	2^{32}-1 bytes
ENUM("value1", …)	65535 members
SET("value1", …)	64 members

CHAR & VARCHAR are the most common string types

CHAR is fixed-width

VARCHAR, BLOBs and TEXTs are variable width

Fixed-length row
- Row containing just fixed length items
- Processed much faster than variable-length rows

MySQL generally converts CHARs to VARCHARS in tables with variable-length rows

BLOB (Binary Large OBject) & Text
- BLOBs use case sensitive comparisons
- TEXT uses case insensitive comparisons

Date & Time Column Types

Type	Range
DATE	"1000-01-01" to "9999-12-31"
TIME	"-835:59:59" to "838:59:59"
DATETIME	"1000-01-01 00:00:00" to "9999-12-31 23:59:59"
TIMESTAMP[(M)]	19700101000000 to year 2037
YEAR[(M)]	1901 to 2155

DATE is time of day

TIME is elapsed time

"12:30" represents "00:12:30"

Basic SQL Commands

```
CREATE TABLE table_name
(
   col_name    col_type [ NOT NULL | PRIMARY KEY]
   [, col_name col_type [ NOT NULL | PRIMARY KEY]]*
)

DROP TABLE table_name

INSERT INTO table_name [(column [, column ]*)]
   VALUES (value [, value]*)

DELETE FROM table_name
   WHERE column OPERATOR value
   [ AND | OR column OPERATOR value ]*

SELECT [table.]column [, [table.]column]*
   FROM table [=alias][, table [= alias]]*
   [ WHERE [table.]column OPERATOR VALUE
   [ AND | OR [table.]column OPERATOR VALUE]*]
   [ ORDER BY [table.]column [DESC][, [table.]column [DESC]]

UPDATE table_name SET column=value [,column=value]*
   WHERE column OPERATOR value
   [ AND | OR column OPERATOR value]*
```

OPERATOR can be <,>,=,<=,>=,<>, or LIKE

VALUE can be a literal value or a column name

Indexing

Column indexes make queries more efficient

MySQL before 3.23.2 did not allow indexed columns to be:

- NULL
- BLOB
- TEXT

Unique & Primary Columns

Unique - index with out duplicate values
Primary key - unique column with index name Primary

Creating Indexes

Can use

- ALTER TABLE
- CREATE INDEX
- CREATE TABLE

Examples - CREATE

Format

```
CREATE TABLE table_name
(
   #create columns, then declare indexes
   INDEX index_name (column_list),
   UNIQUE index_name (column_list),
   PRIMARY KEY (column_list ),
   # more stuff
)

CREATE TABLE roger
(
   sam INT NOT NULL,
   PRIMARY KEY( SAM)
)

CREATE TABLE roger
(
   sam INT NOT NULL PRIMARY KEY
)

CREATE TABLE students
(
   name CHAR(25),
   address CHAR(60),
   INDEX (name, address)
)
```

Alter Table

ALTER TABLE table_name ADD INDEX index_name (column_list)
ALTER TABLE table_name ADD UNIQUE index_name (column_list)
ALTER TABLE table_name ADD PRIMARY KEY (column_list)

Create Index

CREATE UNIQUE INDEX index_name ON table_name (column_list)
CREATE INDEX index_name ON table_name (column_list)

Operators

Arithmetic

+, -, *, / , %

Logical

AND, &&
OR, ||
NOT, !

Bit Operators

&
|
<<
a << b left shift of a by b bits

>> right shift

Comparison Operators

Operator	Example
=	
!=, <>	
<	
<=	
>=	
>	
IN	a IN (x, y, z, …)
BETWEEN	a BETWEEN b AND c
LIKE	a LIKE b
NOT LIKE	
REGEXP, RLIKE	a REGEXP b
NOT REGEXP	
<=>	a <=> b (equal even if NULL)
IS NULL	a IS NULL
IS NOT NULL	

Binary strings
CHAR BINARY, VARCHAR BINARY, and BLOB types

Binary string comparisons are case sensitive

Non-binary string comparisons are not case sensitive

BINARY operator (MySQL 3.23)

Convert a string to binary

BINARY "abc" = "Abc"

Like & Regexp

LIKE patterns match only if the entire string is matched

REGEXP patterns match if the pattern is found anywhere in the string

LIKE is not case sensitive unless at least one operand is a binary string

REGEXP starting in 3.23.4 uses LIKE's case sensitive rules

Like Pattern Matching

Character	Meaning
_	matches any single character
%	matches 0 or more characters of any value
\	escapes special characters

All other characters match themselves

Regexp Pattern Matching

Sequence	Meaning
^	Match the beginning of the string
$	Match the end of string
. (period)	Match any single character
[…]	Match any character between the brackets
[^…]	Match any character not between the brackets
E*	Match zero or more instance of pattern E
E+	Match one or more instance of pattern E
E?	Match zero or one instance of pattern E
E1 | E2	Match E1 or E2
E{m}	Match m instances of E
E{,n}	Match zero to n instances of E
E{m,}	Match m or more instances of E
E{m,n}	Match m to n instances of E
(…)	Group elements in to one element

All other characters match themselves

Expression	Result
"abc" REGEXP "a.c"	1
"abc" REGEXP "[a-z]"	1
"abc" REGEXP "[^a-z]"	0
"abc" REGEXP "^abc$"	1
"abcd" REGEXP "^abc$"	0
"abc" REGEXP "(abc){2}"	0
"abcabc" REGEXP "(abc){2}"	1

Using MySQL In Squeak

You need the MySQL driver and a running version of MySQL

MySQL

MySQL can be downloaded at: http://www.mysql.com/

If you don't want to set up your own MySQL server contact me for a database account on fargo

MySQL Squeak Driver

Part of Comanche 4.5

http://www.eli.sdsu.edu/SmalltalkCode/comanche/index.html

New versions of Comanche do not include the MySQL driver

It is available at:
http://fce.cc.gatech.edu/~bolot/squeak/mysql/Mysql-Driver.28Jan2335.cs.gz

You still need the SocketStream from Comanche

Using MySQL Squeak Driver

The driver supports the following SQL commands:

ALTER	CREATE	DELETE
DROP	GRANT	INSERT
LOCK	REPLACE	SELECT
SET	UNLOCK	UPDATE

Sample Creation of a Table

```
| connection statement resultSet user |
Socket initializeNetwork.
user := JdmConnectionSpec new.
user
        database: '683Examples';
        host: (NetNameResolver addressForName: 'fargo.sdsu.edu');
        port: 5555;
        user: 'cs683';
        password: 'foobar'.
connection := JdmConnection on:  user.
statement := connection createStatement.
statement executeQuery:
   'CREATE TABLE name
      (first CHAR(15),
       second CHAR(15)) '.
statement executeQuery:
   'INSERT INTO  name
   VALUES
      ("Roger" , "Whitney"),
      ("Eli" , "Whitney")'.
resultSet := statement executeQuery:
   'SELECT * FROM name'.
resultSet printString
```

Result

```
 '"first"    "second"
"Roger"  "Whitney"
"Eli" "Whitney"
'
```

Some Explanation

This only needs to be done once after you start an image

```
Socket initializeNetwork.
```

The following gets to be a pain after a while

```
user := JdmConnectionSpec new.
user
    database: '683Examples';
    host: (NetNameResolver addressForName: 'fargo.sdsu.edu');
    port: 5555;
    user: 'cs683';
    password: '******'.
```

I tend to use a subclass

```
JdmConnectionSpec subclass: #CS683ConnectionSpec
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Mysql-Driver'!

initialize
   database := '683Examples'.
   host := NetNameResolver addressForName: 'fargo.sdsu.edu'.
   port := 5555.
   user := 'cs683'.
   password := '******'
```

Class Method

```
new
   ^super new initialize
```

So we then can use:

```
   user := CS683ConnectionSpec new.
```

Statements

In JDBC one uses a statement just once.

Here we can use a statement multiple times

```
connection := JdmConnection on:  user.
statement := connection createStatement.
statement executeQuery:
   'CREATE TABLE name
      (first CHAR(15),
       second CHAR(15)) '.
statement executeQuery:
   'INSERT INTO  name
   VALUES
      ("Roger" , "Whitney"),
      ("Eli" , "Whitney")'.
resultSet := statement executeQuery:
   'SELECT * FROM name'.
```

Return Types of executeQuery

A SELECT query returns a JdmResultSet

All other queries return a JdmResult

JdmResult

Contains type (always update) and number of rows changed

value returns the number of rows changed

JdmResultSet

Important Methods

columns
Returns a collection of JdmColumn objects

next
Gets the next row in the result set from the database
Returns true if the row is not empty

rawValueAt: anInteger
Returns the value of the anInteger column in the row
Value is returned as a string

rawValueNamed: aString
Returns the value of the column with name aString
Value is returned as a string

valueAt: anInteger
Returns the value of the anInteger column in the row
Value is returned as correct Smalltalk type for this column

valueNamed: aString
Returns the value of the column with name aString
Value is returned as correct Smalltalk type for this column

Using the ResultSet

```
| connection statement resultSet user |
Socket initializeNetwork.
user := CS683ConnectionSpec new.
connection := JdmConnection on:  user.
statement := connection createStatement.
resultSet := statement executeQuery: 'SELECT * FROM name'.
columnNames := resultSet columns collect: [:each | each name].
Transcript cr.
columnNames do:
   [:each |
   Transcript
      show: each;
      tab].
[resultSet next]
   whileTrue:
      [Transcript cr.
      columnNames do:
         [:each |
         Transcript
            show: (resultSet valueNamed: each);
            tab.]]
```

Result in Transcript

```
first   second
Roger   Whitney
Eli  Whitney
```

MySQL & Smalltalk Types

All data sent to a MySQL database must be converted to a string

INT Types

valueNamed: valueAt; return an Integer object

```
| connection statement resultSet user anInteger |
Socket initializeNetwork.
user := CS683ConnectionSpec new.
connection := JdmConnection on:  user.
statement := connection createStatement.
statement executeQuery: 'DROP TABLE IF EXISTS typeExamples '.
statement executeQuery: 'CREATE TABLE typeExamples
   (a INT) '.
statement executeQuery: 'INSERT INTO  typeExamples VALUES
   ( 12  ),
   (' , 10 printString , ')'.
resultSet := statement executeQuery:
   'SELECT * FROM typeExamples'.
resultSet next.
anInteger := resultSet valueNamed: 'a'
```

DOUBLE & FLOAT

```
valueNamed: valueAt;
   return a JdmFloatHolder for FLOATs
   return a JdmFloatHolder for DOUBLEs
```

```
| connection statement resultSet user aJdmFloatHolder |
Socket initializeNetwork.
user := CS683ConnectionSpec new.
connection := JdmConnection on:  user.
statement := connection createStatement.
statement executeQuery: 'DROP TABLE IF EXISTS typeExamples '.
statement executeQuery: 'CREATE TABLE typeExamples
   (a FLOAT) '.
statement executeQuery: 'INSERT INTO  typeExamples VALUES
   ( 12.2  ),
   (' , 10.93 printString , ')'.
resultSet := statement executeQuery:
   'SELECT * FROM typeExamples'.
resultSet next.
aJdmFloatHolder := resultSet valueNamed: 'a'.
aFloat := aJdmFloatHolder value.
```

CHAR, VARCHAR, *TEXT

valueNamed: valueAt;
   return a string for CHARs, VARCHARs, *TEXTs

When you send the string in SQL it must be in single quotes.

Note the two different ways to get the quotes in the SQL statement

```
| connection statement resultSet user inputString aString |
Socket initializeNetwork.
user := CS683ConnectionSpec new.
connection := JdmConnection on:  user.
statement := connection createStatement.
statement executeQuery: 'DROP TABLE IF EXISTS typeExamples '.
statement executeQuery: 'CREATE TABLE typeExamples
   (a CHAR(10)) '.
inputString := 'Hi dad'.
statement executeQuery: 'INSERT INTO  typeExamples VALUES
   ( "Hi mom"  ),
   (' , inputString printString ,  ')'.
resultSet := statement executeQuery: 'SELECT * FROM
typeExamples'.
resultSet next.
aString := resultSet valueNamed: 'a'.
```

*BLOB

```
valueNamed: valueAt;
   return a string for all BLOBs

| connection statement resultSet user inputString aByteArray |
Socket initializeNetwork.
user := CS683ConnectionSpec new.
connection := JdmConnection on:  user.
statement := connection createStatement.
statement executeQuery: 'DROP TABLE IF EXISTS typeExamples '.
statement executeQuery: 'CREATE TABLE typeExamples
   (a BLOB) '.
inputString := 'Hi dad'.
statement executeQuery: 'INSERT INTO  typeExamples VALUES
   ( "Hi mom"  ),
   (' , inputString printString ,  ')'.
resultSet := statement executeQuery:
   'SELECT * FROM typeExamples'.
resultSet next.
aByteArray := resultSet valueNamed: 'a'.
aByteArray
```

DATE

valueNamed: valueAt;
   return a JdmDateHolder object

```
| connection statement resultSet user  aJdmDateHolder aDate |
Socket initializeNetwork.
user := CS683ConnectionSpec new.
connection := JdmConnection on:  user.
statement := connection createStatement.
statement executeQuery: 'DROP TABLE IF EXISTS typeExamples '.
statement executeQuery: 'CREATE TABLE typeExamples
   (a DATE) '.
statement executeQuery: 'INSERT INTO  typeExamples VALUES
   ( "2001-05-05"  ),
   (' , Date today yyyymmdd printString ,  ')'.
resultSet := statement executeQuery:
   'SELECT * FROM typeExamples'.
resultSet next; next.
aJdmDateHolder := resultSet valueNamed: 'a'.
aDate := aJdmDateHolder value.
^aDate
```

TIME

valueNamed: valueAt;
return a JdmTimeHolder object

```
| connection statement resultSet user  aJdmTimeHolder aTime |
Socket initializeNetwork.
user := CS683ConnectionSpec new.
connection := JdmConnection on:  user.
statement := connection createStatement.
statement executeQuery: 'DROP TABLE IF EXISTS typeExamples '.
statement executeQuery: 'CREATE TABLE typeExamples
   (a TIME) '.
statement executeQuery: 'INSERT INTO  typeExamples VALUES
   ( "1:3:30"  ),
   (' , Time now print24 printString ,  ')',
   (' , Time now  printString printString ,  ')'
resultSet := statement executeQuery:
   'SELECT * FROM typeExamples'.
resultSet next.
aJdmTimeHolder := resultSet valueNamed: 'a'.
aTime := aJdmTimeHolder value.
^aTime
```

aTime

1:03:30 am

DATETIME

valueNamed: valueAt;
   return a JdmDateTimeHolder object

JdmDateTimeHolder has accessor methods
   date
   time

```
| connection statement resultSet user  aJdmDateTimeHolder  |
Socket initializeNetwork.
user := CS683ConnectionSpec new.
connection := JdmConnection on:  user.
statement := connection createStatement.
statement executeQuery: 'DROP TABLE IF EXISTS typeExamples '.
statement executeQuery: 'CREATE TABLE typeExamples
   (a DATETIME) '.
statement executeQuery: 'INSERT INTO  typeExamples VALUES
   ( "2001-5-5 1:3:30"  )'.
resultSet := statement executeQuery:
   'SELECT * FROM typeExamples'.
resultSet next.
aJdmDateTimeHolder := resultSet valueNamed: 'a'.
```

ENUM

```
valueNamed: valueAt;
   return a string

| connection statement resultSet user  aString  |
Socket initializeNetwork.
user := CS683ConnectionSpec new.
connection := JdmConnection on:  user.
statement := connection createStatement.
statement executeQuery: 'DROP TABLE IF EXISTS typeExamples '.
statement executeQuery: 'CREATE TABLE typeExamples
   (a ENUM( "cat", "dog", "mouse")) '.
statement executeQuery: 'INSERT INTO  typeExamples VALUES
   ( "cat" )'.
resultSet := statement executeQuery:
   'SELECT * FROM typeExamples'.
resultSet next.
aString := resultSet valueNamed: 'a'.
^aString.
```