Database Schema

Revision History

Revision 1.0 November 19, 2003 Revised by: afw
Initial version.
Revision 1.01 November 21, 2003 Revised by: afw
Clarified description of xml_source.
Revision 1.02 December 26, 2003 Revised by: afw
Added discussion of transaction processing.
Revision 1.03 February 3, 2004 Revised by: afw
Various modifications to schema.
Revision 1.04 November 19, 2005 Revised by: jd
More additions to schema.
Revision 1.05 May 9, 2006 Revised by: jd
More additions to schema.
Revision 1.06 June 30, 2008 Revised by: jd
More additions to schema.

The schema for the Spk database (spkdb) is presented in SQL. In addition, the context for the database design is presented, as well as a short description of each table and each field.

Table of Contents

Introduction

This specification describes the logical design of the Spk database. It complements two other specifications:

- Database Entity-Relationship Model [1]
- Job History Model [2]

Architectural Context

The Spk system architecture is comprised of three major independent processes, each of which may have multiple instances running simultaneously. The major processes are the following:

1. *MDA Surrogate.* The Model Design Agent is a client application which runs on the user's workstation. There will typically be many MDAs in operation at any given time. Each MDA communicates with the rest of Spk via a secure Internet connection to a web server. The portion of the web server which supports

the MDA is called the MDA Surrogate. For performance reasons, the architecture permits multiple MDA Surrogates to be running simultaneously on one or more server machines.

2. *Aspk Compiler*. The Application Server for Population Kinetics is a specialized compiler which translates model specifications received from MDAs into C++ source code to be compiled and linked into object code and then run by the Cspk. To optimize performance, there can be multiple Aspk Compilers running simultaneously on one or more server machines.
3. *Cspk*. The Computational Server for Population Kinetics receives source code for a model along with scientific data from the Aspk, utilizes a C++ compiler and linker to build a representation of the model in object code, and then executes it. There can be multiple Cspk server machines and processor clusters.

These three tasks run continuously and, hence, are the type of processes known as *daemons*. A daemon is implemented and installed in such a way that it starts automatically when the host computer on which it runs is booted up, and it stops gracefully when the machine is shut down.

The Spk database is *the sole means of communication* between these daemons. It plays, therefore, a central role in the Spk architecture. The principal interprocess communications are the following:

- *MDA Surrogate to Aspk Compiler*. The MDA submits a user job, via the Surrogate, by creating a new row in the *job* table, storing the model in the *xml_source* field, the data in the *xml_data* field, the user's identification number in the *user_id* field, and setting the *state_code* field to "Queued to Compile". The Aspk Compiler selects the job from the queue when it has become the highest priority job with "Queued to Compile" status, retrieving the model and data and setting *state_code* to "Compiling".
- *Aspk Compiler to MDA*. If, for any reason, the compilation process fails, the Aspk Compiler stores an error report in the *report* field of the job, sets *state_code* to "End" and *end_code* to "Compilation Error". When querying the database, an MDA Surrogate notices that one of its jobs has reached the "End" state and retrieves the report from the database.
- *Aspk Compiler to Cspk*. If the model compiles successfully, the Aspk Compiler stores an archive of C++ source code and data into the *cpp_source* field of the job and sets *state_code* to "Queued to Run". A Cspk selects the job when it is the job of highest priority in the "Queued to Run" state, retrieving the source code and data and setting *state_code* to "Running".
- *Cspk to MDA Surrogate*. When the model runs to completion, the Cspk stores the final report in the *report* field of the job, sets *state_code* to "End" and *end_code* to "Successful Run". When an MDA Surrogate notices that one of its jobs had reached the "End" state, it retrieves the final report.

Several very important points must be emphasized:

- All the high-level interprocess communication described above is *asynchronous*. The loose coupling between the daemons is very attractive from an operational standpoint because additional instances of the daemons can be easily added or subtracted at any time. There is a slight penalty to be paid in terms of communications latency, but this is negligible given the low activity level in the queues.
- Multiple processes may be updating the *job* table simultaneously. These updates must be *atomic* transactions.

Transaction Processing Considerations

In general, multiple independent processes will be modifying the database simultaneously. In designing the database, it is imperative to anticipate situations in which two processes, attempting to do the same thing at the same time, could damage database logical integrity.

As an example, consider the selection of the highest priority job from the compiler queue. There may be several copies of the Aspk Compiler running concurrently, and two of these might attempt to a job from the queue simultaneously. The selection of a job entails the following:

1. Select from the *job* table the job of highest priority which has state_code = 'q2c'.
2. Change the state_code field from 'q2c' to 'cmp', to logically remove the job from the compiler queue.

Both steps must occur as a single transaction, even though time might elapse in between. If this constraint is not enforced the following is one example of what might happen:

1. Compiler-1 selects the highest priority job.
2. Before Compiler-1 can change the state_code in the job, compiler-2 selects the same job.
3. Both Compiler-1 and Compiler-2 compile the job and queue it to be run. This is a waste of resources.
4. Since the job has been queued twice, it runs twice. This is a huge waste of resources and is very likely to confuse the user.

In Version 4 of MySQL, transaction support is provided for tables of type 'InnoDB'. To avoid the problem described above, the *job* table must be created with with 'InnoDB' type, and within programs, the steps required for removing a job from a queue occur between SQL **begin** and **end** statements.

In the Tables and Fields section, transaction processing considerations will be discussed for each table.

SQL Schema

The following SQL statements will create an empty copy of the Spkdb database.

```
CREATE TABLE class (
  class_code char(2) NOT NULL default '',
  class_name char(20) default NULL,
  parent_required tinyint(1) default '0',
  PRIMARY KEY  (class_code)
) TYPE=MyISAM;

CREATE TABLE dataset (
  dataset_id int(10) unsigned NOT NULL auto_increment,
  name varchar(20) NOT NULL default '',
  abstract varchar(100) NOT NULL default '',
  archive longblob,
  user_id int(10) unsigned NOT NULL default '0',
  PRIMARY KEY  (dataset_id),
  UNIQUE KEY user_id (user_id,name)
) TYPE=InnoDB;
```

```
CREATE TABLE end (
  end_code char(4) NOT NULL default '',
  end_name char(20) default NULL,
  PRIMARY KEY  (end_code)
) TYPE=MyISAM;

CREATE TABLE history (
  history_id int(10) unsigned NOT NULL auto_increment,
  event_time int(10) unsigned NOT NULL default '0',
  state_code varchar(4) NOT NULL default '',
  job_id int(10) unsigned NOT NULL default '0',
  host varchar(100) NOT NULL default '',
  PRIMARY KEY  (history_id)
  KEY `idx_history_job_id` (`job_id`)
) TYPE=InnoDB;

CREATE TABLE job (
  job_id int(10) unsigned NOT NULL auto_increment,
  user_id int(10) unsigned NOT NULL default '0',
  abstract varchar(100) NOT NULL default '',
  dataset_id int(10) unsigned NOT NULL default '0',
  dataset_version varchar(10) NOT NULL default '',
  model_id int(10) unsigned NOT NULL default '0',
  model_version varchar(10) NOT NULL default '',
  xml_source longblob,
  checkpoint longblob,
  state_code varchar(4) NOT NULL default '',
  report longblob,
  start_time int(10) unsigned NOT NULL default '0',
  event_time int(10) unsigned NOT NULL default '0',
  cpp_source longblob,
  end_code varchar(4) default NULL,
  method_code char(2) default NULL,
  parent int(10) unsigned default '0',
  mail tinyint(1) default '0',
  share_with int(10) unsigned NOT NULL default '0',
  parallel int(10) NOT NULL default '0',
  folder_id int(10) unsigned NOT NULL default '0',
  PRIMARY KEY  (job_id)
  KEY `idx_job_user_id` (`user_id`)
  KEY `idx_job_share_with` (`share_with`)
) TYPE=InnoDB;

CREATE TABLE method (
  method_code char(2) NOT NULL default '',
  method_name char(20) default NULL,
  class_code char(2) NOT NULL default '',
  test_only tinyint(1) default '0',
  PRIMARY KEY  (method_code)
) TYPE=MyISAM;

CREATE TABLE model (
  model_id int(10) unsigned NOT NULL auto_increment,
  name varchar(20) NOT NULL default '',
  abstract varchar(100) NOT NULL default '',
  archive longblob,
  user_id int(10) unsigned NOT NULL default '0',
  PRIMARY KEY  (model_id),
  UNIQUE KEY user_id (user_id,name)
) TYPE=InnoDB;

CREATE TABLE state (
  state_code char(4) NOT NULL default '',
  state_name char(20) default NULL,
  PRIMARY KEY  (state_code)
) TYPE=MyISAM;
```

```
CREATE TABLE user (
  user_id int(10) unsigned NOT NULL auto_increment,
  first_name varchar(30) NOT NULL default ”,
  surname varchar(40) NOT NULL default ”,
  password varchar(32) NOT NULL default ”,
  username varchar(20) NOT NULL default ”,
  company varchar(30) NOT NULL default ”,
  country varchar(20) NOT NULL default ”,
  state varchar(20) NOT NULL default ”,
  email varchar(40) default NULL,
  test tinyint(1) default ’0’,
  dev tinyint(1) default ’0’,
  team_id int(10) unsigned NOT NULL,
  register_time timestamp default now(),
  contact tinyint(1) NOT NULL default ’1’,
  PRIMARY KEY  (user_id),
  UNIQUE KEY username (username)
) TYPE=InnoDB;

CREATE TABLE team (
  team_id int(10) unsigned NOT NULL auto_increment,
  team_name varchar(20) NOT NULL default ”,
  PRIMARY KEY  (team_id),
  UNIQUE KEY `team_name` (team_name)
) Type=InnoDB;

CREATE TABLE folder (
  folder_id int(10) unsigned NOT NULL default ’0’,
  name varchar(40) NOT NULL default ”,
  parent int(10) unsigned NOT NULL default ’0’,
  user_id int(10) unsigned NOT NULL default ’0’,
  PRIMARY KEY  (user_id,folder_id)
) ENGINE=InnoDB;

DELIMITER ;;
/*!50003 DROP PROCEDURE IF EXISTS `folder_tree` */;;
/*!50003 SET SESSION SQL_MODE=""*/;;
/*!50003 CREATE*/ /*!50020 DEFINER=`root`@`localhost`*/ /*!50003 PROCEDURE `folder_tree
BEGIN
    DECLARE folderId INT;
    DECLARE folderName TEXT;
    DECLARE done INT DEFAULT 0;
    DECLARE cur CURSOR FOR
      SELECT folder_id,name FROM folder WHERE user_id=userId AND parent=parentId;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done=1;

    OPEN cur;
    folder_loop:LOOP

        FETCH cur INTO folderId,folderName;
        IF done=1 THEN
            LEAVE folder_loop;
        END IF;
        SET tree=CONCAT(tree,’<n’,folderId,’ name="’,folderName,’">’);
        CALL folder_tree(userId,folderId,tree);
        SET tree=CONCAT(tree,’</n’,folderId,’>’);
    END LOOP folder_loop;
    CLOSE cur;
END */;;
/*!50003 SET SESSION SQL_MODE=@OLD_SQL_MODE*/;;
DELIMITER ;
```

Static Data

The following SQL statements will populate static data fields in the database:

```
INSERT INTO class (class_code, class_name, parent_required) VALUES ('al','Approx. Likel
INSERT INTO class (class_code, class_name, parent_required) VALUES ('le','Likelihood Ev
INSERT INTO class (class_code, class_name, parent_required) VALUES ('so','Simulation Or
INSERT INTO class (class_code, class_name, parent_required) VALUES ('id','identifiabili

INSERT INTO end (end_code, end_name) VALUES ('cerr','Compiler Error');
INSERT INTO end (end_code, end_name) VALUES ('herr','Hard Fault');
INSERT INTO end (end_code, end_name) VALUES ('serr','Software Error');
INSERT INTO end (end_code, end_name) VALUES ('srun','Successful Run');
INSERT INTO end (end_code, end_name) VALUES ('abrt','User Abort');
INSERT INTO end (end_code, end_name) VALUES ('staf','Opt OK, Stat Failure');
INSERT INTO end (end_code, end_name) VALUES ('othe','Unsuccessful Run');
INSERT INTO end (end_code, end_name) VALUES ('othf','Unknown Run Failure');
INSERT INTO end (end_code, end_name) VALUES ('acce','File Access Error');
INSERT INTO end (end_code, end_name) VALUES ('accf','File Access Failure');
INSERT INTO end (end_code, end_name) VALUES ('sime','Simulation Error');
INSERT INTO end (end_code, end_name) VALUES ('simf','Simulation Failure');
INSERT INTO end (end_code, end_name) VALUES ('opte','Optimization Error');
INSERT INTO end (end_code, end_name) VALUES ('optf','Optimization Failure');
INSERT INTO end (end_code, end_name) VALUES ('stae','Opt OK, Stat Error');
INSERT INTO end (end_code, end_name) VALUES ('usre','Input Error');
INSERT INTO end (end_code, end_name) VALUES ('usrf','Input Failure');
INSERT INTO end (end_code, end_name) VALUES ('deve','Known Prog Error');
INSERT INTO end (end_code, end_name) VALUES ('devf','Known Prog Failure');
INSERT INTO end (end_code, end_name) VALUES ('pose','Post-opt Error');
INSERT INTO end (end_code, end_name) VALUES ('posf','Post-opt Failure');
INSERT INTO end (end_code, end_name) VALUES ('idee','Model Ident Error');
INSERT INTO end (end_code, end_name) VALUES ('idef','Model Ident Failure');
INSERT INTO end (end_code, end_name) VALUES ('spku','SPK Unavailable');
INSERT INTO end (end_code, end_name) VALUES ('rese','Opt OK, Resid Error');
INSERT INTO end (end_code, end_name) VALUES ('resf','Opt OK, Resid Fail');
INSERT INTO end (end_code, end_name) VALUES ('optm','Opt Max Iter Error');

INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('fo','Firs
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('eh','Expe
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('la','Lapl
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('ml','M.C.
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('mc','Mark
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('gr','Gric
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('an','Anal
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('so','Simu
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('ia','Indi
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('mi','Mise
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('s2','Std.
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('i2','Iter
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('g2','Glob
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('sm','MAP
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('im','MAP
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('gm','MAP
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('id','Iden
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('gn','Gric
INSERT INTO method (method_code, method_name, class_code, test_only) VALUES ('un','Unif

INSERT INTO state (state_code, state_name) VALUES ('q2c','Queued to compile');
INSERT INTO state (state_code, state_name) VALUES ('q2ml','Queued to M.L.');
INSERT INTO state (state_code, state_name) VALUES ('cmp','Compiling');
INSERT INTO state (state_code, state_name) VALUES ('q2r','Queued to run');
INSERT INTO state (state_code, state_name) VALUES ('run','Running');
INSERT INTO state (state_code, state_name) VALUES ('end','End');
INSERT INTO state (state_code, state_name) VALUES ('q2ac','Queued to abort cmp');
INSERT INTO state (state_code, state_name) VALUES ('q2ar','Queued to abort run');
INSERT INTO state (state_code, state_name) VALUES ('acmp','Aborting compilation');
```

```
INSERT INTO state (state_code, state_name) VALUES ('arun','Aborting run');
```

Tables and Fields

class

The *class* table categorizes methods into classes. No row is ever deleted from this table.

Table 1. class

Field	Type	Description
class_code	char(2)	Primary key.
class_name	char(20)	Expanded name.
parent_required	tinyint(1)	Whether or not methods in the class can only be used by jobs which have a parent link.

Transaction Processing Considerations

This table is not updated by applications. There are no transaction processing issues.

dataset

Each row of the *dataset* table represents the source code for a scientific dataset. A text file, stored in the *archive* field, contains the current and all previous versions of the dataset, in compressed RCS-compatible format.

Table 2. dataset

Field	Type	Description
dataset_id	int(10) unsigned	Primary key, supplied automatically when the row is created.
name	varchar(20)	The name of the dataset.
abstract	varchar(100)	A short description of the dataset.
archive	longblob	A text file containing the current and all previous versions, in compressed RCS-compatible format.
user_id	int(10) unsigned	The user who owns this dataset. A foreign key to the user table.

Transaction Processing Considerations

The user, with the aid of the MDA, appends rows to this table. The key field, *dataset_id*, is declared with the **auto_increment** modifier and the table has

TYPE=InnoDB, hence MySQL insures that this field is unique.

The application requires each of a user's datasets to have a different name. In order to avoid a potential concurrency problem where two copies of the MDA attempt to add a dataset of the same name for the same user simultaneously, this table must be created with a **UNIQUE KEY user_id (user_id,name)** modifier. With the table thus defined, the responsibility for avoiding this race condition is assumed by MySQL, and applications do not have to be concerned with it.

end

The *end* table provides short and long text for the end condition. No row is ever deleted from this table.

Table 3. end

Field	Type	Description
end_code	char(4)	Primary key.
end_name	char(20)	Expanded name.

Transaction Processing Considerations

This table is not updated by applications. There are no transaction processing issues.

history

The *history* table records all changes of state of every job. Each time that the *state_code* field of a row in the *job* table is changed, a row must be added to *history*. Rows are never deleted from this table.

Table 4. history

Field	Type	Description
history_id	int(10) unsigned	Primary key. A unique positive integer, supplied automatically when a row is created.
event_time	int(10) unsigned	The time at which the row was added to the table. This value is the number of seconds since the year 1970 began, at the prime meridian.
state_code	varchar(4)	The state to which the given job transitioned at the given time. A foreign key to the state table.
job_id	int(10) unsigned	The job that underwent a transition at the given time. A foreign key to the job table.
host	varchar(100)	The host name of the computer on which the job was running at the time the event occurred.

Transaction Processing Considerations

Applications append rows to this table. Once in the table, a row is never updated. The value of the key field, *history_id*, is provided automatically by MySQL. There are

no transaction processing considerations.

folder

Each row in this table represents a folder that contains jobs for user.

Table 5. folder

Field	Type	Description
folder_id	int(10) unsigned	A unique positive integer, folder_id for a user.
user_id	int(10) unsigned	The user to whom the job belongs. A foreign key to the user table.
name	varchar(40)	The name of the folder.
parent	int(10) unsigned	The folder_id of the parent folder of this folder.

Transaction Processing Considerations

There are no transaction processing considerations.

job

Each row in this table represents an Spk job. It is updated each time the job makes a state transition. Rows are never deleted from this table.

Table 6. job

Field	Type	Description
job_id	int(10) unsigned	Primary key. A unique positive number, supplied automatically when the row is created.
user_id	int (10)	The user to whom the job belongs. A foreign key to the user table.
abstract	varchar(100)	Short description of the job.
dataset_id	int(10)	The dataset, a version of which this job runs. A foreign key to the dataset table.
dataset_version	varchar(10)	The rcs version code of the dataset being run.
model_id	int(10)	The model, a version of which this job runs. A foreign key to the model table.
model_version	varchar(10)	The rcs version code of the model being run.
xml_source	longblob	An XML file containing source code for the model, constraints, parameters and presentation directives that is provided by the MDA when the row is created. This file is input to the Aspk Compiler, which translates it into the cpp_source field, for use by the Cspk.

Field	Type	Description
checkpoint	longblob	An XML file containing checkpoint information that was written to a file by the job after its last iteration. The runtime daemon copies the file to the database.
state_code	varchar(4)	The current state of the job. A foreign key to the state table.
report	longblob	The final report, expressed as XML, added to the row at the time the job makes the transition to the "End" state.
start_time	int(10) unsigned	The time, to a resolution of one second, at which the job was first submitted to Spk. The value stored is the number of seconds since the year 1970 began, at the prime meridian.
event_time	int(10) unsigned	The time, to a resolution of one second, at which the job entered its current state. The value stored is the number of seconds since the year 1970 began, at the prime meridian.
cpp_source	longblob	An archive of files, containing c++ source code and scientific data, which is output by the Aspk Compiler for the use of the Cspk when the job enters the "Queued to Run" state.
end_code	varchar(4)	When the job reaches the 'end' state, the end_code is set to indicate the type of completion that occurred. A foreign key to the end table.
method_code	char(2)	The method to be used for the calculation. A foreign key to the method table.
parent	int(10)	The job_id of the parent job, if this job depends on a previous one. A foreign key to the job table.
mail	tinyInt(1)	Whether or not the user wants an end-job email notice.
share_with	int(10)	The user_id of the user whom the job is shared with.
parallel	int(10)	The number of sub-tasks if the job runs in parallel processing mode. Being zero if the job runs in single process mode.
folder_id	int(10)	The folder_id of the folder containing the job.

Transaction Processing Considerations

Applications append rows to this table. This action does not have any transaction processing implications, because the key field, *job_id*, is provided automatically by MySQL, as long as *job_id* is declared with the **auto_increment** modifier and the table has **TYPE=InnoDB**.

On the other hand, several Aspk Compilers may attempt to update the same row of the table simultaneously, in the act of selecting a job to be compiled. In order to maintain the integrity of the table, the code that performs queue selection and the associated update of the *state_code* field must executed in transaction state. In other words, a SQL **begin** statement or its programming language equivalent, must be executed before the queue selection code starts, and a **commit** command must be

executed afterward.

Similarly, several Cspk process may attempt simultaneously to select the highest priority job that is ready to run. The situation is fully equivalent to that of selecting a job to compile, as described above, and the remedy is the same, as well.

method

The *method* table provides short and long text for the method of calculation. No row is ever deleted from this table.

Table 7. method

Field	Type	Description
method_code	char(2)	Primary key.
method_name	char(20)	Expanded name.
test_only	tiny_int(1)	Whether or not this method can only be employed by a user qualified by a true value in the test column of his or her row in the user table.

Transaction Processing Considerations

This table is not updated by applications. There are no transaction processing issues.

model

Each row of the *model* table represents the source code for a scientific model. A text file, stored in the *archive* field, contains the current and all previous versions of the model, in compressed RCS-compatible format.

Table 8. model

Field	Type	Description
model_id	int(10) unsigned	Primary key, supplied automatically when the row is created.
name	varchar(20)	The name of the model.
abstract	varchar(100)	A short description of the model.
archive	longblob	A text file containing the current and all previous versions, in compressed RCS-compatible format.
user_id	int(10) unsigned	The user who owns this model. A foreign key to the user table.

Transaction Processing Considerations

The user, with the aid of the MDA, appends rows to this table. The key field, *model_id*, is declared with the **auto_increment** modifier and the table has

TYPE=InnoDB, hence MySQL insures that this field is unique.

The application requires each of a user's models to have a different name. In order to avoid a potential concurrency problem where two copies of the MDA attempt to add a model of the same name for the same user simultaneously, this table must be created with a **UNIQUE KEY user_id (user_id,name)** modifier. With the table thus defined, the responsibility for avoiding this race condition is assumed by MySQL, and applications do not have to be concerned with it.

state

The *state* table provides short and long names for the states that a job can be in. Rows must never be deleted from this table.

Table 9. state

Field	Type	Description
state_code	char(3)	Primary key. The short name of a job state.
state_name	char(20)	Long name for a state.

Transaction Processing Considerations

This table is not updated by applications. There are no transaction processing issues.

user

The *user* table contains information about the user. The MDA consults this table when a user logs in. This table identifies jobs and models. Rows must never be deleted from this table.

Table 10. user

Field	Type	Description
user_id	int(10) unsigned	Primary key. Provided automatically when the row is created.
first_name	varchar(30)	The given name of this user.
surname	varchar(40)	The family name of this user.
password	varchar(32)	An encrypted password, used when logging in.
username	varchar(20)	A user name used for logging in. Must be unique.
company	varchar(30)	The name of the company the user works for.
country	varchar(20)	The user's country.
state	varchar(20)	The user's country or province.
email	varchar(40)	The user's electronic mail address.

Field	Type	Description
test	tinyint(1)	Whether or not the user is authorized to access features that are still under test.
dev	tinyint(1)	Whether the user is one of the developers of this software.
team_id	int(10) unsigned	The team_id of the user, 0 if the user is not in a team.
register_time	timestamp	The time when the user is registered.
contact	tinyint(1)	Whether the user is in the contact list.

Transaction Processing Considerations

Users are added by an administrative application. The key field, *user_id*, is automatically assigned to be unique by MySQL, as long as the **auto_increment** modifier is used and the table has **TYPE=InnoDB** when the table is created. This insures that there are no concurrency issues when adding a user.

Once a user is in the database, she can use an MDA to modify her record. Because the *username* field must be unique within the table, there could be a race condition with two users trying to change to the same username. Problems are avoided by specifying **UNIQUE KEY (username)** and **TYPE=InnoDB**.

team

The *team* table contains information about the team. The MDA consults this table when a user logs in. This table identifies users of the team. Rows must never be deleted from this table.

Table 11. team

Field	Type	Description
team_id	int(10) unsigned	Primary key. Provided automatically when the row is created.
team_name	int(10) unsigned	The given name of this team.

Transaction Processing Considerations

Teams are added by an administrative application. The key field, *team_id*, is automatically assigned to be unique by MySQL, as long as the **auto_increment** modifier is used and the table has **TYPE=InnoDB** when the table is created. This insures that there are no concurrency issues when adding a team.

Once a team is in the database, users can join the team. A team can have many users, but a user can join only one team.

Notes

1. ../erModel/erModel.html
2. ../jobHistory/jobHistory.html