Bachelor's thesis

Degree programme: Information Technology

2012

Elena Oat

REPORTING AND FUNCTIONALITY ADDITION FOR TIME-TRACKING SYSTEM

TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR´S THESIS | ABSTRACT
TURKU UNIVERSITY OF APPLIED SCIENCES

Degree programme | Information Technology

23/05/2012| 50 pages

Instructors: Jaakko Väliviita , Patric Granholm

Elena Oat

REPORTING AND FUNCTIONALITY ADDITION FOR TIME-TRACKING SYSTEM

The purpose of this study was to implement additional features to a configured time-tracking system used in a company. The desired functionality included automatic clock-out of the employees and reporting features. The main goal of the thesis consisted of creating an “interface” between end user application and database. Therefore, execution of SQL queries from application object-oriented code was avoided. Transact-SQL stored procedures written for Microsoft SQL Server 2008 R2 constitute components of the “interface”. Procedures were written using the set-based approach which ensures conciseness of code and fast processing time.

KEYWORDS:

Transact SQL, stored procedure

CONTENT

APPENDICES

TABLES

1 INTRODUCTION

1.1 Problem definition

The profit and success of a company depends directly on how efficiently it operates. One of the premises of efficient work lies in task automation, so that machines execute work they are able to deal with correctly, while humans configure and observe machines. Naturally not in every task people can be replaced by machines, but in those cases when applicable, the efficiency of work grows significantly, while the amount of errors and costs goes down. Most of the companies nowadays have implemented some method of tracking the working hours of their employees. Paychecks are issued according to the acquired data, which ensures that employees are paid in accordance with their completed work. Some employers still use pen and paper to take notes on when employees come to work and when they leave. As a result, managers have piles of paper on their table by the end of the month. The consequences of this type of approach include considerable amount of hours spent on manual calculations as well as human errors.

RFID technology helps alleviating this problem. Employees would have to wave their own RFID tag in front of the reader when they come to work and wave it again when leaving. As each tag has a unique number assigned by its manufacturer, this ensures that each employee receives individual tag that is tied to their name. An administrator configures the system so that data is inserted into the right location. After this, the system is automated. A large number of companies use this type of system to keep track of working hours, as this removes the overhead of manual tasks.

1.2 Project goals

The purpose of the project described in this thesis is to improve an existing clock-in system and provide additional functionality to it. The company where the system is used is a small one, where the number of employees does not exceed 15. Nevertheless, the owner took the decision to implement a time-tracking system. This proves that implementation costs are justified even for the case of a small organization.

The goals set in this study include design and implementation of additional features that would make the system more practical and automate larger amount of tasks. One of the problems posed is clocking out employees who have forgotten to do so. Later they should be able to correct the timestamp entered automatically by the system. The administrator of the system should know later whether a stamp was inserted manually by employee or automatically by a script, therefore, the stamps need to have a mark that makes them distinguishable from the manually entered stamps.

To keep the database entries intact, data entered by the script in an automatic manner should remain in the database, while timestamps with accurate time of leave will be inserted into an auxiliary table and will be tied with a foreign key to data in the main timestamp table.

Another problem that requires a solution is the ability of monitoring corrections on automatic timestamps. The manager of the company should be able to see whether all employees have corrected automatically inserted timestamps. This will ensure accurate paychecks.

Reporting on worked hours represents another set goal for the thesis. Managers, as well as regular employees, need to see summaries of the amount of worked hours. Employers' reports should include information on all employees, while others should see data related only to them.

The implementation of desired functionality for clock-in system is done using Transact-SQL. Stored procedures written in Transact-SQL solve most of the required features. Simple script in Ruby language, that makes use of a Ruby library 'Tiny_tds' for database connection, executes corresponding stored procedure designed for automatic clock-out. To schedule Ruby script, a scheduled task in Task Scheduler, that is included in Windows OS, is configured to run the script at desired time of specified days.

Stored procedures are chosen as a solution to keep GUI code separate from SQL. This allows modifying and improving Transact-SQL code if needed, while not making radical changes to the GUI code. Additionally, focus on SQL as an individual part of the project, results usually in a better written and more efficient code.

The scope of the thesis does not include developing an application that would use T-SQL code. Application development and GUI coding represents another half of the project of clock-in/out system improvement. A software developer, who works for the company, will work on application development after the SQL procedures are written, and solutions for set requirements are approved.

Briefly summarizing, the goals of the thesis are:

- Developing a mechanism of automatic clock-out
- Developing a solution for correcting automatically inserted timestamps with accurate ones and keeping data integrity at the same time
- Delivering a reporting system on the amount of worked hours.

1.3 Concepts and definitions

The following section lists all definitions and concepts that relate to the thesis. The purpose of this section is to clarify and define terms used throughout the work so that the reader could refer to those if needed.

Only selected definitions are provided to keep this work concise.

1.3.1 Definitions

SQL stored procedure is a subroutine that accesses a relational database. Its purpose is to provide execution result of some logical operations, included in the procedure, to applications. Stored procedure or sproc, as it is called sometimes for brevity, is stored in a database itself. [1]

RFID (Radio Frequency Identification) is a technology of non-contact transfer of data. A typical case of use would be transfer of unique identification number from an RFID tag to a reader. Radio-frequency electromagnetic fields are used to transfer data. RFID technology is mostly used for automatic tracking and identification.

Transact-SQL or T-SQL is Microsoft’s extension to SQL. T-SQL is an expansion on SQL to allow use of local variables, procedural programming, changes to DELETE and UPDATE statements, etc. T-SQL provides richer possibilities for writing code. [2]

SQL join is an operation applied on tables or queries that combines records from these tables or queries that follow a rule specified in the join definition.

SQL inner join is a type of join when tables are joined on the basis of a comparison operator. [3]

While inner join returns only records from joined tables when the comparison condition is met for both of the tables, **left outer join or left join** returns all records (all that satisfy conditions specified in WHERE and HAVING) from the first participant table (the one that is being immediately followed by FROM clause) in the join operation. [4]

SQL self join is a type of SQL join where a table is joined to itself. [5]

Self, inner and left outer joins are implemented in almost all stored procedures that apply to this project.

Ruby is an open source object oriented programming language that promotes writing simple and productive code. [6]

1.3.2 Concepts

This section presents the main concepts that were applied in this thesis work.

A) RFID technology: applications and characteristics

Radio frequency identification is a technology that has become significantly more common in the recent years due to the significant cost drop and benefits that come with its implementation. Most people in developed countries apply it in their daily life in transportation, libraries, work, etc.

RFID has several advantages over other technologies of identification: it can identify objects from a distance, it does not require the object to be uncovered, the object can, in fact, be in a box. These two features are not specific to barcodes, however. RFID tags support a larger amount of unique identifiers, which means that a larger amount of objects can be identified by it. They can also store additional information in their memory: their manufacturer, product type, etc. Hundreds of RFIDs can be identified by a reader at the same time, while barcodes can be read only one by one. [7]

RFID is being used in a variety of ways. Thanks to its miniature size, it can be inserted under the skin of animals to track their movements. They are even being implanted into human bodies in rare cases, when, for example, police should keep track of police officers that participate in risky operations. At the same time, they are widely used in retail chains to track products and prevent their theft. One of the common uses of RFID are card-shaped tags that are used for access to buildings, time and attendance tracking of employees. [8] [9]

Characteristics

RFID tags can be of passive or active type. The differentiating feature between them is their source of power: passive tags do not have internal power supply, they are being “charged” by the reader in their proximity, whereas active tags are equipped with a battery, which allows their identification from larger distances. Both types of tags have their advantages and disadvantages and should be chosen in accordance with their purpose. Absence of power supply in passive tags defines their small sizes, while active tags are known for their accuracy, reliability and capability of being read from many meters away.

Card-shaped and black leather key-fobs tags are being used for time-tracking purposes in the company for which this thesis is written. Passive tags were chosen because of their relatively low cost and provided functionality. It is unnecessary to identify people from long distances in this particular case.

Privacy and security concerns

While being an extremely beneficial and promising technology, RFID poses security problems too. Illegal tracking of people or objects represents one of the possible threats. Their location can be identified by readers from considerable distances, which makes them exposed to a variety of threats. Concurrently, private information stored in the memory of RFID tags can be read by an intruder. [9]

Another security concern is the possibility of cloning RFID tags, which means that anyone could access private property. An intruder could use a fake access card, that simulates a genuine one, and get hold of private information or goods.

All possible risks should be taken into consideration and prevention rules should be set when technology is being implemented in an organization.

B) SQL programming challenges

At first sight, SQL appears as a simple and trivial language. That’s why not many people take time to master it and consider knowledge of SELECT and a couple of other statements as being enough to achieve most of the desired functionality. It is worth mentioning that almost any organization nowadays uses databases for daily tasks. Therefore, the efficiency and reliability of database systems is closely connected with their operation. A multitude of concepts and best practices are known that help administrators and programmers develop well-operating environments. Yet even experienced DB programmers learn and discover new methods all the time, that replace previously considered best practices. This process is continuously ongoing.

Below are presented some of the concepts that were implemented while working on this thesis.

Set-based versus procedural approach

Most SQL programmers are aware of the fact that SQL Server does not work best with row by row operations, while it performs in a most efficient manner with sets. Thus, two types of coding approaches are distinguished when speaking about subroutines in Transact-SQL.

The procedural approach represents a programmatic approach, where a programmer tells the system what should be calculated and in which way results should be calculated. In other words, the programmer makes decisions on methods and procedures of result calculation.

While the procedural approach promotes decision making by the programmer, the main idea of the set-based approach is to let the system decide on the methods of how to compute results. The programmer’s task in the set-based approach consists in telling the system what should be calculated, and not how it should be calculated.

The SQL server execution engine was designed so that it operates on sets of data and this ensures its best performance and fast execution results. When programmers tell the engine how to execute operations it has to take suggested path instead of using the best and optimized solution that it can decide on its own. This way programmers are not doing any favor to SQL operating efficiency, on the contrary, they are depriving it of its right of using the most efficient and fastest method of calculating results.

A perfect example of the set-oriented approach advantage over the procedural one are joining techniques. Three types of them can be distinguished: merge join, nested loop join, hash join. Resourcewise, their cost can be characterized as the least for merge join and the highest for hash join. These joins are each applicable for different situations, depending on the number of records and indexes. When a join operation is implemented in the SQL code, the programmer does not specify which type of join will be used. SQL Server determines which of them three is applicable and would supply the best results. [10]

Time-based data

Most databases store and manage time-based data or, in different terms, temporal data. The domain of values for this data includes working periods, time when an employee comes to work or leaves and the total amount of worked hours. All mentioned values are characteristic and essential for this thesis.

In spite of requirements set by life, databases can store only discrete time values, i.e., only instants in time. This poses challenges when dealing with time-related values. Because time is continuous and time values can be represented with different approximation, strict rules should be considered when developing and managing databases. Any later additions or modifications related to records or database structure have to follow defined rules. This ensures consistency and accuracy, as well as simplifies work performed by administrators and programmers.

The main concepts related to time, that are also addressed when creating set of rules for database management, are presented below:

- *Granularity*

The granularity concept represents the smallest unit of time that is used in a database. Granularity can be, for example, up to a minute or a second, depending on the data that is stored and set rules. Granularity should be defined clearly when performing any database operations. In other words, granularity represents the smallest time unit that the organization cares about.

- *Precision*

Precision, on the other hand, is defined by the datatype of data. For example DATETIME datatype has a precision of 3.3 milliseconds in SQL Server. As a result, any DATETIME value, that is entered into database, has milliseconds specified too, even if they are not provided when the data is inserted.

- *Duration*

Duration, in terms of SQL, means an interval in time and is, usually, an integer.

Instance, in SQL, is a moment in time.

The ISO 8601 standard is chosen in this work as a way of representing dates and times, hence their format of representation. It specifies that all input and output date and time values in SQL code are of format "YYYY-MM-DD HH:MM:SS.MMM". [12]

The implementation section of this work presents challenging situations encountered in the project related to time concepts in databases.

2 OVERVIEW AND DETAILS OF THE CURRENT CLOCK-IN SYSTEM

The researched system represents a working-time recording model for company employees. The system includes several elements that offer required functionality:

- RFID tags and readers, which provide data to database about employee clock-in and clock-out time;

- Ruby daemon running on the DB server that inputs data to database about events of clock-in and clock-out;

- Database that holds information about employees, tags associated with employees and clock-in and clock-out stamps.

Further are described technical specifications of the above mentioned system components. Additionally, the purpose and the description for each element are provided to clarify the concepts further.

Tags

RFID tags used in the company are passive Mifare tags, which can hold information of 1 kilobyte. Each RFID tag has its own unique number of 4 bytes. Thus to each employee corresponds a unique number which is stored in the tag memory. This number is also stored in the database in the table which holds information about employees.

Readers

Access 7CE readers manufactured by Idesco are implemented in the company system for collecting data supplied by RFID readers. These readers are able to read unique ID number of Philips Mifare tags. There are two readers in total, one by the entrance door and another one next to dining room, so employees could comfortably clock their time.

Each reader has its own IP address and is connected directly through Ethernet cable to company network. Power to readers is supplied by the Ethernet network.

Reader has a small LED as well that lights up when an event occurs. When a tag is in reader’s proximity, LED flashes briefly red and then either continues flashing red or turns green for couple of seconds, which means that the user has been clocked out or clocked in, correspondingly. The buzz is also switched on when any of mentioned above events occur. It emits a short sound which indicates that the user is either clocked in or out. The type of event is evident from the color of the flashing LED.

Ruby daemon

Daemon, running on the database machine, serves as a middleware between the readers and database. Its main purpose is to catch events which are recorded by the reader and insert a tag’s unique id, as well as toggle the status of the employee from present to absent or vice versa in the corresponding table. Database is updated in the following way with latest events concerning employees that have been either clocked in or clocked out.

Ruby daemon is just a small program that runs two threads in this case, as there are two readers. Each of the threads has an observable and observers. The observable’s purpose is to recognize when a change occurs, i.e., when data is read by the reader, and notify this change to the observers that in turn take relevant actions. In our case, an observer is used that checks whether the data read by the reader is valid, i.e., whether the tag exists and is related to an employee in the company. In case the tag is valid, the observer updates the data in the database, toggling the status of the employee whose tag was read by the reader. Otherwise, an error message is displayed by daemon and changes are not implemented.

Ruby daemon has another important role, too. It notifies employees whether they have been clocked in or clocked out. The event type can be recognized by the LED color: if it is red, the user has been clocked out, and if green, the user has been clocked in.

Database

The researched system's database server runs on the Windows 2008 R2 Standard edition. The type of DMBS used for managing data is relational – RDBMS. Microsoft SQL Server was chosen as the database server software.

The database engine product version is Microsoft SQL Server 2008 Service Pack 1 and it is a Standard 64-bit Edition.

The database server supports mixed authentication mode, thus it allows to log in to SQL server using SQL Server authentication and Windows authentication. SQL Server authentication is used by the ruby daemon to get access to database and modify its records.

Only one database is used for managing all the data supplied by the readers.

Table 1 illustrates the database schema.

3 REQUIREMENTS AND DESIRED FEATURES OF THE IMPROVED CLOCK-IN SYSTEM

This section of thesis describes the desired functionality of the system, as well as tasks and requirements that are implemented in order to achieve this functionality. The described tasks represent, at the same time, the goals of the thesis.

Below are presented the missing features of the system that need to be implemented in the current system implementation:

A. Automatically clock-out employees who have not been clocked out from the system by the end of the day.

B. Allow employees to correct later the time of automatic clock-out.

C. Create a reporting system that would show the worked hours of employees.

By solving the above presented problems, the DBA (database administrator) would not have to worry about deleting records from the tables that have incorrect clock-out time, as well as inserting by hand correct replacements.

At the same time, employees will not have to worry all the time that they forget to clock-out, because they will have to go through the process of asking the administrator to delete their incorrect stamps and insert new ones.

If the number of employees is fewer than ten, manual management of the system is possible, but what if the company employs hundreds of workers? In this case, the DBA would have to spend half of his day by fixing consequences of careless attitude towards the system. Automatic clock-out will also allow to avoid human error when inserting and deleting data to database.

Another problem set for this work, which represents a significant part of the thesis, is reporting. The goal is to allow employees to see hours worked during a period of time. At the moment, employees are not able to verify how many hours they have done so far. Besides, the manager of the company should be able to check hours worked by all employees for a specified period. This way, the employer could pay employees according to worked hours and write accurate paychecks.

It is worth mentioning that reports should include correct clock-out time for all employees. Thus, before counting the paycheck, all automatically clocked out employees should have corrected their time accordingly. For this purpose another report should be created that lists all employees and dates when they have been clocked out automatically and whose timestamps have not been corrected.

Additionally, reports should include data on the hours worked till moment when the query is run. In other words, if employees would like to know how many hours they worked during the current month, the report should include worked hours for that current day, too. In case the employee has not been clocked out, a temporary clock-out is set in the procedure that does not affect the table data, which equals to current time. In this way the employee receives the most accurate report on worked hours.

Below follows the breakdown of the steps necessary to satisfy the requested features:

A., B. Automatic clock-out

I. Modifying the existing database structure to fit desired functionality and features (refer to Table 2 to view the updated DB design).

II. Designing an automatic system that would insert automatically records for employees and update their status:

- Creating a stored procedure that would search for employees that have not been clocked out by the end of the day and update status, insert clock-out timestamp and a record for future timestamp correction for them

- Creating a ruby script that would run the above mentioned procedure
- Creating a scheduled task for the script.

III. Creating a stored procedure that would allow employees to correct automatically inserted values for clock-out time.

IV. Creating a stored procedure that would show the employees that have been clocked out automatically and have not corrected the automatically inserted timestamp.

V. Revising ruby daemon to avoid insertion of incorrect data.

C. Reporting

I. Writing T-SQL code for each report type.

Reports should provide information to an employee/manager on the following:

- Hours worked on specific day for one/all employee(s)
- Hours worked in a specific week for one/all employee(s)
- Hours worked during a month for one/all employee(s)
- Hours worked for a free range of days for one/all employee(s).

4 SOLUTIONS FOR SET GOALS AND IMPLEMENTATION

4.1 Implementation challenges and their solutions

This subsection lists challenging situations during the implementation phase. The problem description, as well as the solution to it, is presented here.

Set-oriented thinking against procedural thinking

It is essential to realize the power of set-based approach in Transact-SQL, i.e., thinking of work with a set of records at once, instead of thinking of work with one record at a time. Execution times of stored procedures and queries written in set-oriented style are considerably faster than once written in a procedural way. Another great advantage is the conciseness of code written in set-oriented style. Usually queries are much shorter and easier to grasp, than the ones written in procedural style, where code is long. This presented one of the significant challenges the author had to deal with during implementation.

One example is provided for comparison in Appendix 11.

Procedures in Appendices 6 and 11 implement the same functionality – they calculate the amount of hours worked by employees during one day. The same task is being approached from set-based (Appendix 6) and procedural (Appendix 11) points of view. As it can be noticed, the procedure is 4 pages long and contains a large amount of local variables, which represents a typical feature of procedural SQL programming. The execution time of this procedure is several times slower than of its alternative written using the set-oriented approach.

Concept of DATETIME datatype

When users say they want to find out how many hours have they worked during a specified day, they mean a duration - from morning till evening of that day. When a user queries a database for a day "2011-11-11", SQL server casts the given value to a DATETIME and it becomes an instant that equals to "2011-11-11 12:00:00.000". In other words, it is the same if the user would ask how long he has worked on 11th of November 2011 at midnight. Any DATETIME value, in fact, represents a millisecond in time.

It is also important to note that strings given in a form 'YYYY-MM-DD' are being implicitly casted by the server into DATETIME values in Transact-SQL.

Additional code needs to be written to calculate correctly required values and this concept needs to be fully understood to evaluate correctly what is being asked. [11]

Calculation of duration and its human-readable representation

The work in the thesis is closely tied with time and duration. Most of the data stored in the researched database are of type DATETIME.

Calculating the duration of an event presents a challenge, as duration is not of type DATETIME, though it is calculated based on the values of type DATETIME.

Granularity presents a great importance in calculation of durations. In this work, granularity is set to seconds. So all durations are presented as an INTEGER, which implies the duration in seconds of a happening. Because thousands of seconds is not the most readable way of presenting duration, the integer number is converted when needed into a varchar variable in the form "hh:mm:ss".

The DATEDIFF function is largely used in procedures written for the thesis. It allows calculating the difference between two instants given as timestamps in seconds. [12]

NULL results

Sometimes when running a query null results are returned. For example calculating the SUM of values in a field on an empty table will return null. At the same time, the GROUP BY clause applied on an empty table will return empty result set. As a convention for this project, an empty result set will be returned instead of NULL to keep consistency among all procedures.

Local temporary tables and their scope

Local temporary tables are used in almost all of the stored procedures written for this project. Their main purpose is to store temporarily the result set of a complex query.

Although local temporary tables are explicitly created with CREATE TABLE #TName statement, they do not have to be dropped explicitly, too. The table is dropped implicitly when the stored procedure ends. That is why the statement DROP TABLE #TName is not found in the procedures.

Calculating hours for employees currently at work

Employees that have not been clocked out, because they are currently at work, cannot see the accurate amount of worked hours in case they query a period that includes the current day. To overcome this limitation, a temporary stamp needs to be inserted that simulates a clock-out. This allows calculating correctly hours worked by an employee including the current day. The concept of a dual table is used to accomplish this task in most RDBMS. In T-SQL, the dual table does not exist. Neither is needed, as "FROM" is not a required component of the SELECT statement in T-SQL.

4.2 Implementation

4.2.1 Automatic clock-out feature

I. Database schema modification

Two changes are required for existing database to fit in desired functionality:

1. Creating table “corrections” that is meant to hold information about manually corrected employees’ clock-out time. A foreign key is defined on the table which references the id of a stamp from the “stamps” table. Thus each row in “corrections” table relates to a row in “stamps” table by its “id_stamps” field value.

2. Adding a new field of type bit to table “stamps” which will indicate whether the entered stamp was inserted automatically by scheduled task, whose purpose is to automatically clock-out all employees. The default value of the field is 0, which indicates that the stamp was not inserted automatically by scheduled job.

II. Automatic update of database records

a. In order to insert and update records from the database with the automatic clock-out information, a new stored procedure should be created that would execute this task.

The procedure performs the following tasks:

- Checks all employees that have a status 1 in the people table (those who are not clocked out)
- Inserts into the stamps table a record that contains a current timestamp for corresponding employees
- Inserts into the corrections table a record that references just inserted into the stamps record, that will be used later for inserting the accurate time of clock-out by the employee
- Updates the people table and sets the status to 0 for the

corresponding employees.

The code of the procedure can be seen in Appendix 1.

b. To clock out employees automatically, a scheduled job should be created using Task Scheduler. This scheduled job will run ruby script that runs with the above mentioned stored procedure.

Ruby script uses Tiny_tds library to connect to the database. It checks whether the connection is active and if it is, it executes the procedure. Its code can be found in Appendix 2.

c. The at command is used to schedule a periodic run of the ruby script.

The following command creates a scheduled task that runs ruby script every workday at 23:30

at 23:30 /every: M, T, W, Th, F ruby “C:\Documents and Settings\administrator\Jobs\cron.rb”

In our case the ruby executable path is added to the Windows environment variables.

The Task Scheduler service is running on the server system that allows to schedule tasks.

The job is run under the System account, which has a few advantages over running the task under some other account. For example, in case the password expires for a usual account, the script will cease to run until the password is updated. Using a system account overcomes this limitation.

III. Correct automatically inserted timestamp

Created for correcting automatically inserted timestamps, the SQL stored procedure takes two parameters: employee id and correct timestamp, i.e., the date and time when an employee left the work place on a corresponding day.

Procedure searches for all automatically inserted timestamps on the respective day for a specified employee, and then updates the table with an accurate timestamp.

The source code for procedure can be found in Appendix 3.

IV. Report on employees who have not corrected automatically inserted timestamps

Report represents a stored procedure that shows a complete list of employees and dates, when they have not corrected yet their automatically inserted clock out time.

The manager, who is responsible for paychecks, will have to run this report before creating payrolls for employees.

Source code for procedure is provided in Appendix 4.

V. Implementation of required changes to daemon

As the ruby daemon does not directly store any information about employees or their timestamps, but pools this info from the database, it can be presumed independent from the changes that happen in the database (tables are updated with new data from other sources). In other words, by inserting new data to database, daemon’s functionality will not be modified anyhow. Although a new table will be created and a field will be added to an existing table, this will not affect the operation of the ruby program, as no data will be pulled or inserted from the new table. Regarding the new field, when data is inserted by the daemon to the corresponding table, the value for the new column is inserted by the database itself, as there is a default value defined.

4.2.2 Reporting feature

The purpose of this feature consists of creating two types of reports: employee reports, that would show the worked hours for a day, month or free range period for specified employee, and employer reports that provide summary of worked hours by all company employees during one day, month or free range period.

Six different procedures were written to satisfy the desired requirements. All of them use the set-based approach, which allows executing them faster and in an efficient manner, with least resource expenses.

The source code and comments for procedures can be found in Appendices 5-10.

5. CONCLUSION AND FUTURE WORK

The purpose of the thesis was to enhance the existing time-tracking system with additional features to make it more practical for use. All desired functionalities were implemented although some limitations still exist. As an example, local temporary tables creation in stored procedures should be, preferably, avoided in the future. It is worth mentioning that even though local temporary tables appear in the procedures' code, their use is optimized – only relevant rows and columns are inserted into temporary table from the original one, instead of the whole original table.

Stored procedures' usage can be all together skipped in some cases.

The latest SQL Server release, SQL Server 2012, brings with it new features and additions, including two analytic functions: LEAD and LAG. [14] Using these functions would simplify code for most stored procedures written for this project. Specifically, the usage of one self join could be avoided, which would result in a shorter and clearer code.

All listed improvements require additional time for development, testing, comparison and enhancement.

The set-oriented approach was used for developing Transact-SQL code for an "interface" between database and application. It presents the most efficient and concise style of SQL programming. This allows easier reading of the code by other developers and fast processing times of the code.

The development of graphical user interface for the application that will use Transact-SQL stored procedures follows after this project is finished. This will ensure full report functionality of the time-tracking system.

SOURCE MATERIAL

[1] Stored procedure basics. MSDN library. Consulted 15.12.2011

http://msdn.microsoft.com/en-us/library/ms190782%28v=sql.105%29

[2] Transact-SQL. Consulted 03.03.2012

http://en.wikipedia.org/wiki/Transact-SQL

[3] Using inner joins. MSDN library. Consulted 05.04.2012

http://msdn.microsoft.com/en-us/library/ms190014%28v=sql.105%29

[4] Using outer joins. MSDN library. Consulted 10.04.2012

http://msdn.microsoft.com/en-us/library/ms187518%28v=sql.105%29

[5] Using joins. MSDN library. Consulted 11.03.2012

http://msdn.microsoft.com/en-us/library/ms177490%28v=sql.105%29.aspx

[6] Ruby language. Consulted 12.12.2011

http://www.ruby-lang.org/en/

[7] Want, R. 2006. An introduction to RFID technology. PERVASIVE computing, 22-33. Consulted 10.05.2012

http://w.thispervasiveday.com/documents/articles-perspectives/an-introduction-to-rfid-technology.pdf

[8] What can RFID be used for? Consulted 11.05.2012

http://www.technovelgy.com/ct/Technology-Article.asp?ArtNum=4

[9] RFID tags - Radio Frequency Identification tags. Consulted 11.05.2012 http://www.rfident.org/

[10] Shubho, A. 2009. Understanding “Set-based” and “Procedural” approaches in SQL. Consulted 03.03.2012

http://www.codeproject.com/Articles/34142/Understanding-Set-based-and-Procedural-approaches

[11] Špetič, A.; Gennick J., 2002. Transact-SQL Cookbook. Sebastopol, CA: O’Reilly.

[12] Celko, J. 2011. SQL for smarties. Advanced SQL Programming. Fourth Edition. Elsevier.

[13] Sorrells, P. 1998. Passive RFID Basics. Microchip Technology Inc., 1-4. Consulted 10.12.2011http://ecee.colorado.edu/~ecen4021/notes/Modulation.pdf

[14] Analytic functions. MSDN library. Consulted 05.05.2012

http://msdn.microsoft.com/en-us/library/hh213234

Automatical clock-out of employees SQL stored procedure

```sql
USE [tikuTesti]
GO
/****** Object:  StoredProcedure [dbo].[ClockOutDS2]    Script Date:
04/30/2012 13:35:24 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =============================================
-- Description:     Clock-out employees who forgot
-- to clock-out
-- =============================================
ALTER PROCEDURE [dbo].[ClockOutDS2]

AS
BEGIN
     SET NOCOUNT ON;


     -- table variable
     DECLARE @correction_id table
     (id int)

     INSERT INTO stamps(stype, person_id, automatic)
     OUTPUT inserted.id INTO @correction_id
     SELECT 0, id, 1
     FROM people
     WHERE status = 1


     INSERT INTO corrections(id_stamps)
     SELECT id FROM @correction_id


     UPDATE people
     SET status = 0
     WHERE status = 1


END
```

Ruby script that runs stored procedure of automatic clock-out

```
===========================================================
require "rubygems"
require "tiny_tds"

client = TinyTds::Client.new(:dataserver=>'DBSERVER',
:database =>'TikuDB', :username=>'tikuadmin',
:password=>'p@ssw0rd')

if client.active?
begin
     result = client.execute("exec ClockOut")
     result.each
end


end
```

Appendix 3

Correction of automatic timestamps SQL stored procedure

```
USE [tikuTesti]
GO
/****** Object:  StoredProcedure [dbo].[CorrectedTime]    Script Date:
04/30/2012 13:34:31 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =============================================
-- Description:      This procedure will insert correct time for
employee who has
-- been clocked out automatically

-- User enters the person_id and correct date and time
-- by entering date the date when correction should be entered is
known
-- by entering time the correct time is know
-- =============================================
ALTER PROCEDURE [dbo].[CorrectedTime]
     @person_id int,
     @date datetime

AS
BEGIN


     SET NOCOUNT ON;

     UPDATE corrections
     SET time = @date
     WHERE id_stamps =
     (SELECT c.id_stamps from corrections c
     JOIN
     stamps s
     ON c.id_stamps = s.id
     WHERE s.person_id = @person_id
     and  convert(varchar(10),  s.time,  121)  =  convert(varchar(10),
@date, 121))


END
```

Report on automatically inserted timestamps that were not corrected SQL stored procedure

```
USE [tikuTesti]
GO
/****** Object:  StoredProcedure [dbo].[NoCorrectedTime]    Script
Date: 04/30/2012 13:33:12 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =============================================
-- Description: This procedure displays
-- list of employees who have not corrected
-- automatically inserted timestamp with real
-- time of their clock-out time FROM work
-- =============================================
ALTER PROCEDURE [dbo].[NoCorrectedTime]
AS
BEGIN
     SET NOCOUNT ON;

     -- first join matches all employee ids with names
     SELECT name AS [Employee name], t.time AS [Automatic stamp] FROM
people
     INNER JOIN
     (SELECT stamps.person_id, stamps.time  FROM stamps
     -- second join matches all employee ids that have corrected time
null
     INNER JOIN corrections
     on stamps.id = corrections.id_stamps
     WHERE corrections.time is NULL) t
     on t.person_id = people.id
     ORDER BY [Automatic stamp]

END
```

Report on hours worked by an employee in a day SQL stored procedure

```
USE [tikuTesti]
GO
/****** Object:  StoredProcedure [dbo].[HoursInDay]    Script Date: 
05/10/2012 17:32:59 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =============================================
-- Description: This procedure calculates
-- amount of hours worked by an employee
-- =============================================
ALTER PROCEDURE [dbo].[HoursInDay]
 @work_day datetime,
 @person_id int
AS
BEGIN
-- SET NOCOUNT ON;

 -- temp table
 CREATE TABLE #stamps
 (time_stamp datetime, stype int)

 INSERT INTO #stamps(time_stamp, stype)
 SELECT
  ISNULL(c.time, s.time),
  stype FROM stamps AS s
 LEFT JOIN
 (
  SELECT
   time,
   id_stamps
  FROM corrections
  WHERE corrections.time is NOT NULL
 ) AS c
 ON s.id = c.id_stamps
 WHERE s.person_id = @person_id
 AND CONVERT(varchar(10), s.time, 121) = CONVERT(varchar(10),
@work_day, 121)

 INSERT INTO #stamps
 SELECT GETDATE(), 0
 WHERE CONVERT(VARCHAR(10), @work_day, 121) = CONVERT(VARCHAR(10),
GETDATE(), 121)
 AND @person_id IN (SELECT id FROM people WHERE status = 1)


 SELECT
  CONVERT(VARCHAR(10), s1.time_stamp, 121) AS [Work Day],
  SUM(DATEDIFF(second, s1.time_stamp, s2.time_stamp)) AS [Seconds],
  RIGHT(ltrim(str(SUM(DATEDIFF(second, s1.time_stamp,
s2.time_stamp))/3600)), 3) + ':' +
  RIGHT('0' + ltrim(str(SUM(DATEDIFF(second, s1.time_stamp,
s2.time_stamp))%3600/60)), 2) + ':' +
  RIGHT('0' + ltrim(str(SUM(DATEDIFF(second, s1.time_stamp,
s2.time_stamp))%3600%60)), 2) AS [Hours/HumanReadable]
 FROM
 (
  SELECT
   time_stamp, stype,
   RANK() OVER(ORDER BY time_stamp) AS
   RankNr
  FROM #stamps
 ) AS s1
 JOIN
 (
  SELECT
   time_stamp, stype,
   RANK() OVER(ORDER BY time_stamp) AS
   RankNr
  FROM #stamps
 ) AS s2
 ON
 s1.RankNr = s2.RankNr - 1
 AND s1.stype = 1
 AND s2.stype = 0
 GROUP BY CONVERT(VARCHAR(10), s1.time_stamp, 121)

END
```

Report on hours worked by all employees in a day SQL stored procedure

```
USE [tikuTesti]
GO
/****** Object:  StoredProcedure [dbo].[HoursDayAll]    Script Date: 
05/10/2012 17:34:23 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =============================================
-- Description: This procedure calculates the 
-- hours worked for all employees during one day
-- day is entered as a parameter
-- =============================================
ALTER PROCEDURE [dbo].[HoursDayAll]
 @work_day AS DATETIME
AS
BEGIN
 SET NOCOUNT ON;

 --temp TABLE
 CREATE TABLE #stamps
 (person_id int, time_stamp DATETIME, stype int)

 INSERT INTO #stamps(person_id, time_stamp, stype)
 SELECT
  s.person_id,
  ISNULL(c.time, s.time) as [time stamp],
  s.stype
 FROM stamps AS s
 LEFT JOIN
 (SELECT * FROM corrections
 WHERE corrections.time is not null) as c
 ON s.id = c.id_stamps
 WHERE CONVERT(varchar(10), s.time, 121) = CONVERT(VARCHAR(10), 
@work_day, 121)


 INSERT INTO #stamps(person_id, time_stamp, stype)
 SELECT id, GETDATE(), 0
 FROM PEOPLE
 WHERE status = 1
 AND CONVERT(varchar(10), @work_day, 121) = CONVERT(VARCHAR(10), 
GETDATE(), 121)


 SELECT
  p.name AS [Employee Name],
  q.[Employee ID] AS [Employee ID],
  q.[Seconds] AS [Seconds],
  q.[Hours/HumanReadable] as [Hours/HumanReadable]
 FROM people as p
 JOIN
 (
  SELECT
   s1.person_id AS [Employee ID],
   ISNULL(SUM(DATEDIFF(ss, s1.time_stamp, s2.time_stamp)), 0) AS
[Seconds],
   RIGHT ('00' + ltrim(str(SUM(DATEDIFF(second, s1.time_stamp,
s2.time_stamp))/3600)), 2) + ':' +
   RIGHT ('00' + ltrim(str(SUM(DATEDIFF(second, s1.time_stamp,
s2.time_stamp))%3600/60)), 2 )+ ':' +
   RIGHT ('00' + ltrim(str(SUM(DATEDIFF(second, s1.time_stamp,
s2.time_stamp))%3600%60)), 2) AS [Hours/HumanReadable]
  FROM
  (
   SELECT
    person_id,
    time_stamp,
    stype,
    RANK() OVER(PARTITION BY person_id ORDER BY time_stamp) AS RankNr
    FROM #stamps
  ) AS s1
  JOIN
  (
  SELECT
   person_id,
   time_stamp,
   stype,
   RANK() OVER (PARTITION BY person_id ORDER BY time_stamp) AS RankNr
  FROM #stamps
  ) AS s2
  ON s2.RankNr = s1.RankNr + 1
  WHERE s1.person_id = s2.person_id
  AND s1.stype = 1
  AND s2.stype = 0
  GROUP BY s1.person_id
 ) AS q
 ON q.[Employee ID] = p.id


END
```

Report on hours worked by an employee during a month SQL stored procedure

```
USE [tikuTesti]
GO
/****** Object:  StoredProcedure [dbo].[HoursInMonth]    Script Date: 05/10/2012 17:35:08 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =============================================
-- Description: This procedure calculates 
-- amount of hours worked by an employee 
-- during a month
-- =============================================
ALTER PROCEDURE [dbo].[HoursInMonth]
 @person_id int,
 @month_nr int,
 @year_nr int
AS
BEGIN
 SET NOCOUNT ON;

 CREATE TABLE #stamps
 (time_stamp datetime, stype int)

 INSERT INTO #stamps(time_stamp, stype)
 SELECT
  ISNULL(c.time, s.time) AS [stamp],
  s.stype AS [stype]
 FROM stamps s
 LEFT JOIN
 corrections AS c
 ON
 s.id = c.id_stamps
 WHERE s.person_id = @person_id
 AND DATEPART(month, s.time) = @month_nr
 AND DATEPART(year, s.time) = @year_nr

 INSERT INTO #stamps(time_stamp, stype)
 SELECT GETDATE(), 0
 WHERE DATEPART(MONTH, GETDATE()) = @month_nr
 AND DATEPART(YEAR, GETDATE()) = @year_nr
 AND @person_id IN
 (SELECT id FROM people WHERE status = 1)
 SELECT
  SUM(DATEDIFF(second, s1.time_stamp, s2.time_stamp))AS [Seconds],
  RIGHT(ltrim(str(SUM(DATEDIFF(second, s1.time_stamp,
s2.time_stamp))/3600)), 3) + ':' +
  RIGHT ('0' + ltrim(str(SUM(DATEDIFF(second, s1.time_stamp,
s2.time_stamp))%3600/60)), 2)+ ':' +
  RIGHT ('0' + ltrim(str(SUM(DATEDIFF(second, s1.time_stamp,
s2.time_stamp))%3600%60)), 2) AS [Hours/HumanReadable]
 FROM

  (SELECT
   time_stamp,
   stype,
   RANK() OVER(ORDER BY time_stamp) AS RankNr
   FROM #stamps
  )AS s1
  JOIN
  (
   SELECT
    time_stamp,
    stype,
    RANK() OVER(ORDER BY time_stamp) AS RankNr
   FROM #stamps
  )AS s2
  ON s1.RankNr = s2.RankNr - 1
  WHERE s1.stype = 1
  AND s2.stype = 0
  AND DATEPART(DAY, s1.time_stamp) = DATEPART(DAY, s2.time_stamp)
  GROUP BY DATEPART(MONTH, s1.time_stamp)

END
```

Appendix 8

Report on hours worked by all employees during a month SQL stored procedure

```sql
USE [tikuTesti]
GO
/****** Object:  StoredProcedure [dbo].[HoursMonthAll]    Script Date: 
05/10/2012 17:36:20 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =============================================
-- Description: This procedure calculates hours
-- for all employees during a month
-- =============================================
ALTER PROCEDURE [dbo].[HoursMonthAll]
 @month_nr int,
 @year_nr int
AS
BEGIN
 SET NOCOUNT ON;

 CREATE TABLE #stamps
 (time_stamp datetime, person_id int, stype int)

 INSERT INTO #stamps(time_stamp, person_id, stype)
 SELECT ISNULL(c.time, s.time) AS [stamp], s.person_id AS [personID],
s.stype AS [stype] FROM stamps s
 LEFT JOIN
 corrections AS c
 ON
 s.id = c.id_stamps
 WHERE DATEPART(month, s.time) = @month_nr
 AND DATEPART(year, s.time) = @year_nr


 INSERT INTO #stamps(time_stamp, person_id, stype)
 SELECT GETDATE(), p1.id, 0
 FROM people AS p1
 JOIN
 (SELECT id FROM people
 WHERE status = 1 )AS p2
 ON p1.id = p2.id
 AND DATEPART(MONTH, GETDATE()) = @month_nr


 SELECT
  p.name AS [Employee Name],
  p.id AS [Employee ID],
  k.[Seconds] AS [Seconds],
  k.[Hours Worked] AS [Hours/HumanReadable]
 FROM people AS p
 JOIN
 (SELECT
  s1.person_id AS [Employee ID],
  SUM(DATEDIFF(SECOND, s1.time_stamp, s2.time_stamp)) AS [Seconds],
  RIGHT(ltrim(str(SUM(DATEDIFF(SECOND, s1.time_stamp,
s2.time_stamp))/3600)), 3) + ':' +
  RIGHT('0' + ltrim(str(SUM(DATEDIFF(SECOND, s1.time_stamp,
s2.time_stamp))%3600/60)), 2) + ':' +
  RIGHT('0' + ltrim(str(SUM(DATEDIFF(SECOND, s1.time_stamp,
s2.time_stamp))%3600%60)), 2) AS [Hours worked]

 FROM
  (SELECT
   time_stamp,
   person_id,
   stype,
   RANK() OVER(PARTITION BY person_id ORDER BY time_stamp) AS RankNr
  FROM #stamps) AS s1

  JOIN
  (
   SELECT
    time_stamp,
    person_id,
    stype,
    RANK() OVER(PARTITION BY person_id ORDER BY time_stamp) AS RankNr
   FROM #stamps
  )AS s2

  ON
  s1.RankNr = s2.RankNr - 1
  WHERE s1.person_id = s2.person_id
  AND s1.stype = 1
  AND s2.stype = 0
  AND DATEPART(DAY, s1.time_stamp) = DATEPART(DAY, s2.time_stamp)
 GROUP BY s1.person_id
 ) AS k
 ON
 k.[Employee ID] = p.id


END
```

Report on hours worked by an employee during a free range period SQL stored procedure

```
USE [tikuTesti]
GO
/****** Object:  StoredProcedure [dbo].[HoursInPeriod]    Script Date:
05/10/2012 17:37:00 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =============================================
-- Description: This procedure calculates the
-- amount of hours worked during a given period
-- of time for a specific employee
-- =============================================


ALTER PROCEDURE [dbo].[HoursInPeriod]
@start datetime,
@end datetime,
@person_id int
AS
BEGIN
 SET NOCOUNT ON;

 -- temp table
 CREATE TABLE #stamps
 (time_stamp DATETIME, stype int)

 INSERT INTO #stamps
 SELECT
  ISNULL(c.time, s.time),
  s.stype
 FROM stamps AS s
 LEFT JOIN
 corrections AS c
 ON s.id = c.id_stamps
 WHERE person_id = @person_id
 AND CONVERT(VARCHAR(10), s.time, 121) >= CONVERT(VARCHAR(10), @start,
121)
 AND CONVERT(VARCHAR(10), s.time, 121) <= CONVERT(VARCHAR(10), @end,
121)



 INSERT INTO #stamps
 SELECT GETDATE(), 0
 WHERE CONVERT(VARCHAR(10), GETDATE(), 121)
 BETWEEN CONVERT(VARCHAR(10), @start, 121)
 AND CONVERT(VARCHAR(10), @end, 121)
 AND @person_id IN (SELECT id FROM people WHERE status = 1)

 --select * from #stamps

 SELECT
  SUM(DATEDIFF(SECOND, s1.time_stamp, s2.time_stamp)) AS [Seconds],
  RIGHT(LTRIM(STR(SUM(DATEDIFF(SECOND, s1.time_stamp,
s2.time_stamp))/3600)), 3) + ':' +
  RIGHT ('0' + LTRIM(STR(SUM(DATEDIFF(SECOND, s1.time_stamp,
s2.time_stamp))%3600/60)), 2)+ ':' +
  RIGHT ('0' + LTRIM(STR(SUM(DATEDIFF(SECOND, s1.time_stamp,
s2.time_stamp))%3600%60)), 2) AS [Hours worked]
 FROM
 ((SELECT
  time_stamp,
  stype,
  RANK() OVER (ORDER BY time_stamp) AS RankNr
 FROM #stamps) AS s1
 JOIN
 (SELECT
  time_stamp,
  stype,
  RANK() OVER (ORDER BY time_stamp) AS RankNr
 FROM #stamps) AS s2
 ON
 s1.RankNr = s2.RankNr - 1
 AND s1.stype = 1
 AND s2.stype = 0
 AND DATEPART(DAY, s1.time_stamp) = DATEPART(DAY, s2.time_stamp))
 HAVING COUNT(*) > 0


END
```

Appendix 10

Report on hours worked by all employees during a free range period SQL stored procedure

```
USE [tikuTesti]
GO
/****** Object:  StoredProcedure [dbo].[HoursPeriodAll]    Script
Date: 05/10/2012 17:37:46 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =============================================
-- Description: This procedure calculates the
-- amount of hours worked during a given period
-- of time for all employees
-- =============================================


ALTER PROCEDURE [dbo].[HoursPeriodAll]
@start DATETIME,
@end DATETIME

AS
BEGIN
 SET NOCOUNT ON;



 -- temp table
 CREATE TABLE #stamps
 (person_id int, time_stamp DATETIME, stype int)



 INSERT INTO #stamps
 SELECT
  s.person_id,
  ISNULL(c.time, s.time),
  s.stype
 FROM stamps AS s
 LEFT JOIN
 corrections AS c
 ON s.id = c.id_stamps
 WHERE CONVERT(varchar(10), s.time, 121) >= CONVERT(varchar(10),
@start, 121)
 AND CONVERT(varchar(10), s.time, 121) <= CONVERT(varchar(10), @end,
121)

 INSERT INTO #stamps
 SELECT
  id, GETDATE(), 0
 FROM people AS p
 WHERE status = 1
 AND CONVERT(VARCHAR(10), GETDATE(), 121) BETWEEN CONVERT(varchar(10),
@start, 121)
 AND CONVERT(varchar(10), @end, 121)

SELECT
  p.name AS [Employee Name],
  q.[Employee ID] AS [Employee ID],
  q.[Seconds] AS [Seconds],
  q.[Hours/HumanReadable] as [Hours/HumanReadable]
 FROM people as p
 JOIN

 (SELECT
  s1.person_id AS [Employee ID],
  SUM(DATEDIFF(SECOND, s1.time_stamp, s2.time_stamp)) AS [Seconds],
  RIGHT(LTRIM(STR(SUM(DATEDIFF(SECOND, s1.time_stamp,
s2.time_stamp))/3600)), 3) + ':' +
  RIGHT ('0' + LTRIM(STR(SUM(DATEDIFF(SECOND, s1.time_stamp,
s2.time_stamp))%3600/60)), 2)+ ':' +
  RIGHT ('0' + LTRIM(STR(SUM(DATEDIFF(SECOND, s1.time_stamp,
s2.time_stamp))%3600%60)), 2) AS [Hours/HumanReadable]

 FROM
 (SELECT
  person_id,
  time_stamp,
  stype,
  RANK() OVER(PARTITION BY person_id ORDER BY time_stamp) AS RankNr
 FROM #stamps) AS s1
 JOIN
 (SELECT
  person_id,
  time_stamp,
  stype,
  RANK() OVER(PARTITION BY person_id ORDER BY time_stamp) AS RankNr
 FROM #stamps) AS s2
 ON
 s1.RankNr = s2.RankNr - 1
 AND s1.person_id = s2.person_id
 AND s1.stype = 1
 AND s2.stype = 0
 AND CONVERT(VARCHAR(10), s1.time_stamp, 121) = CONVERT(VARCHAR(10),
s2.time_stamp, 121)
 GROUP BY s1.person_id ) AS q
 ON p.id = q.[Employee ID]
END
```

Appendix 11

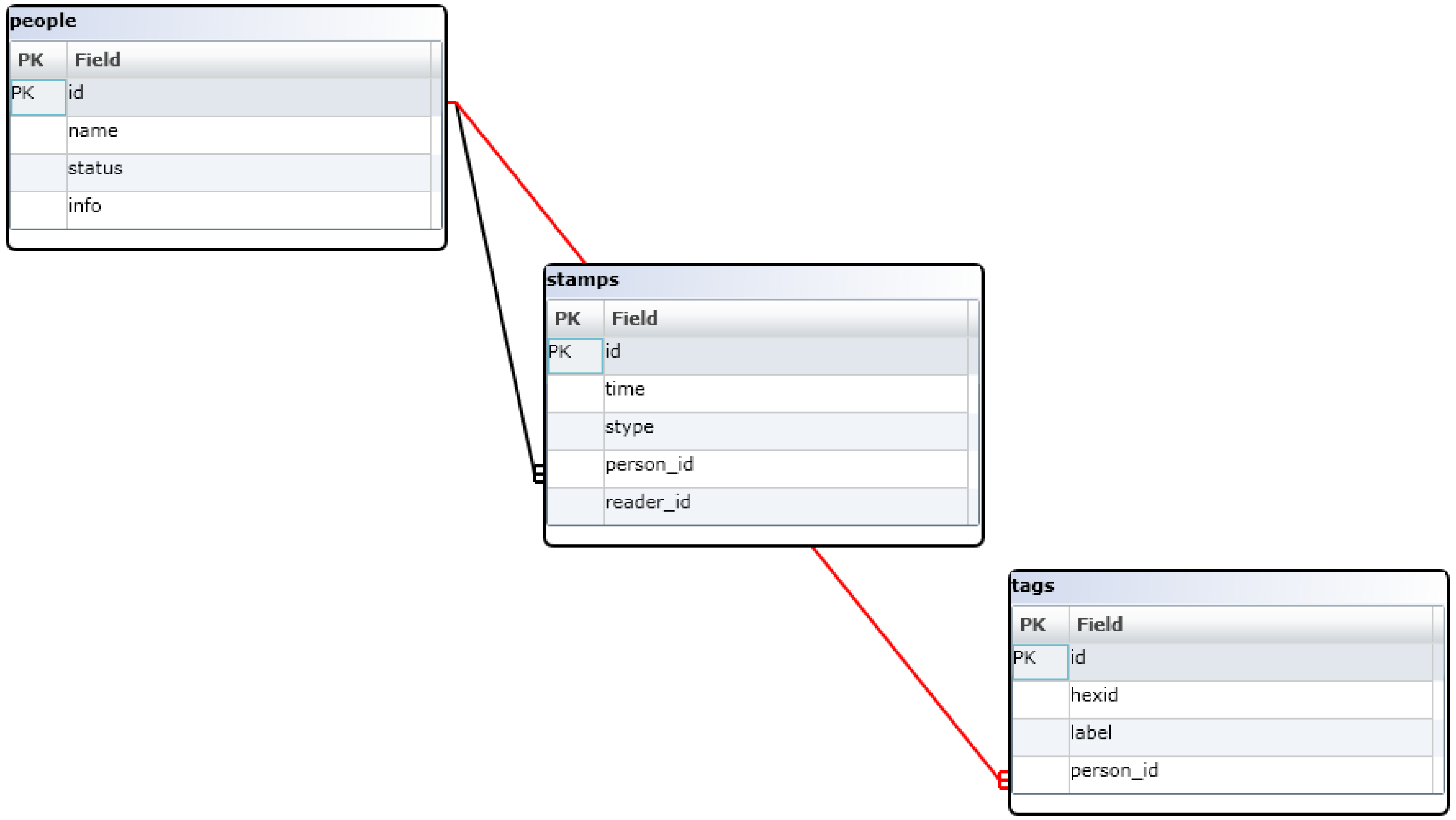
SQL stored procedure written using procedural approach. Procedure calculates the amount of hours worked by all employees

```
USE [tikuTesti]
GO
/****** Object:  StoredProcedure [dbo].[StampsToday]    Script Date:
04/30/2012 16:04:17 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =============================================
-- Description: This procedure calculates
-- amount of hours worked during
-- that day by each employee
-- =============================================
ALTER PROCEDURE [dbo].[StampsToday]
     -- Add the parameters for the stored procedure here
     @today datetime,
     @result bit
AS
BEGIN
     SET NOCOUNT ON;

     select GETDATE()
     declare @time as datetime
     declare @name as varchar(50)
     declare @stype as int
     declare @automatic as bit
     declare @stamp_out as datetime
     declare @min as datetime
     declare @id as int

     -- vars for comparing nr of in and out stamps
     declare @stamps_in_nr as int
     declare @stamps_out_nr as int


     declare @h as int
     declare @m as int
     declare @s as int

     select @today = dbo.dateonly(@today)

     create table #stamps (id int, name varchar(50), time datetime,
stype int, automatic bit)

     insert into #stamps
     select id, dbo.employeename(person_id), time, stype, automatic
from stamps
     where dbo.dateonly(time) = @today


     declare unstamped cursor
     for select distinct name from #stamps

     open unstamped
     fetch next from unstamped into @name

     while @@fetch_status = 0
     begin
          select @stamps_in_nr = (select count(id) from #stamps where
stype = 1 and name = @name)
          select @stamps_out_nr = (select count(id) from #stamps where
stype = 0 and name = @name)
          if @stamps_in_nr > @stamps_out_nr
          begin
               -- if today = today's date, employees are temp clocked
out
              if @today = dbo.dateonly(getdate())
               begin
                    insert #stamps(name, time, stype, automatic)
                    (select @name, getdate(), 0, 0)
               end
               else
               begin
                    print 'Clock-out stamps for employee ' + @name
                    + ' are missing.' + char(13) +
                    'Run Clock-out procedure for ' +
convert(varchar(10), @today, 121) + ' to clock-out everyone.' +
                    char(13) + 'Exiting the code.'
                    close unstamped
                    deallocate unstamped
                    return 1
               end
          end
          fetch next from unstamped into @name
     end

     close unstamped
     deallocate unstamped

     create table #stampstoday(name varchar(50), date varchar(10),
time_in varchar(15),
     time_out varchar(15), hours_worked varchar(15))


     declare stamps_in cursor
     for select name, time from #stamps
     where stype = 1

     open stamps_in
     fetch next from stamps_in
     into @name, @time

     while @@fetch_status = 0
     begin
```

```
          declare stamps_out cursor
          for select id, time, automatic from #stamps
          where stype = 0 and name = @name


          open stamps_out
          fetch next from stamps_out into @id, @stamp_out, @automatic

          -- min initial value = next day
          select @min = @today + 1
          while @@fetch_status = 0
          begin
               if @automatic = 1
               begin
                    if (select time from corrections where id_stamps =
@id) > 0
                    begin
                         select @stamp_out = (select time from
corrections where id_stamps=@id)

                    end
               end
               if ( datediff(millisecond, @time, @stamp_out) > 0) and
               (datediff(millisecond, @stamp_out, @min) > 0)
               begin
                    select @min = @stamp_out
               end
               fetch next from stamps_out into @id, @stamp_out,
@automatic
          end
          close stamps_out
          deallocate stamps_out

          insert into #stampstoday
          select @name, convert(varchar(10), @today, 121),
convert(varchar(15), @time, 114), convert(varchar(15), @min, 114),
convert(varchar(15), (@min - @time), 114)
          fetch next from stamps_in into @name, @time
     end

     close stamps_in
     deallocate stamps_in


     -- having data for all stamps for a day, calculate total time
worked for this day for each user

     create table #totalhours(name varchar(50), date varchar(10),
total varchar(15))

     declare @hours as varchar(15)
     declare @sum as varchar(15)
     declare total cursor
     for select distinct name
     from #stampstoday

     open total
     fetch next from total into @name
     while @@fetch_status = 0
     begin
          declare sumhours cursor
          for select hours_worked from #stampstoday
          where name = @name

          select @sum = '00:00:00'
          open sumhours
          fetch next from sumhours into @hours

          while @@fetch_status = 0
          begin
               select @h=h, @m=m, @s=s from
dbo.calculatesumtime(@hours, @sum)
               select @sum = dbo.convertedtime_str(@h, @m, @s)
               fetch next from sumhours into @hours
          end
          close sumhours
          deallocate sumhours
          insert into #totalhours
          values(@name, convert(varchar(10), @today, 121), @sum)
          fetch next from total into @name
     end

     close total
     deallocate total

     -- return result set if @result param equals 1
     if @result = 1
     select * from #totalhours

     create table #timecount(workday varchar(10), employee
varchar(50), workedhours varchar(15))

     if (select count(*) from timecount) != 0
     begin

       insert #timecount(workday, employee, workedhours)
       select workday, employee, workedhours from timecount
       where workday = convert(varchar(10), @today, 121)
       if (select count(*) from #timecount) != 0
       begin
          print 'There is already some data for this day in timecount
table'
       end


     end



     select getdate()

END
```

Table 1

Original database structure of clock-in system



people

PK	Field
PK	id
	name
	status
	info

stamps

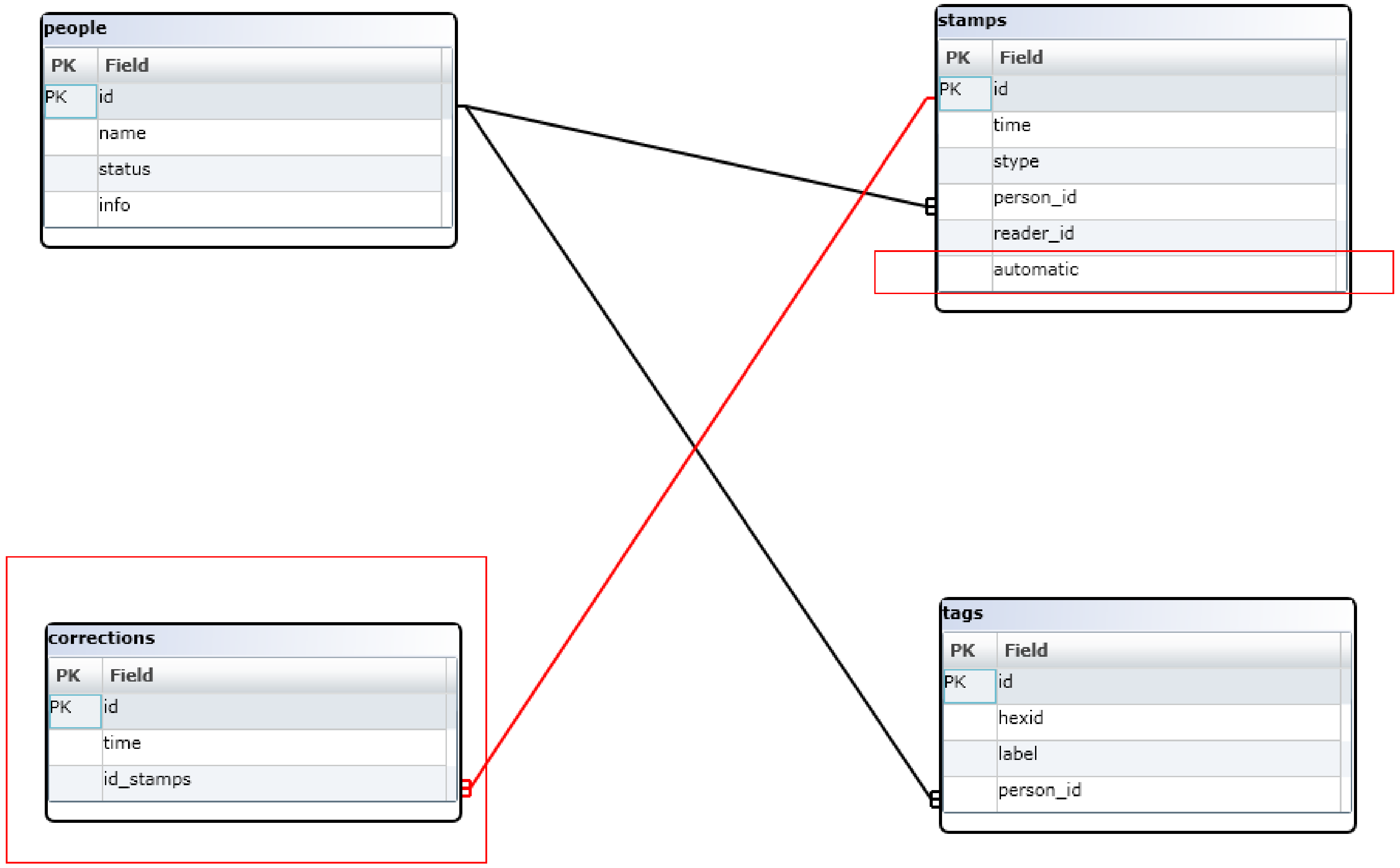
PK	Field
PK	id
	time
	stype
	person_id
	reader_id

tags

PK	Field
PK	id
	hexid
	label
	person_id

Table 2

Database schema of improved clock-in system



people

PK	Field
PK	id
	name
	status
	info

stamps

PK	Field
PK	id
	time
	stype
	person_id
	reader_id
	automatic

corrections

PK	Field
PK	id
	time
	id_stamps

tags

PK	Field
PK	id
	hexid
	label
	person_id