SC15068

Learning problem solving with Excel, Access and MySQL

Jerzy Letkowski
Western New England University

Nicholas Letkowski
Independent Consultant

Abstract

Popular software applications taught in business curricula include spreadsheet and database technologies. Typical settings for learning processes arrange spreadsheet driven problem solving separately from database oriented problem solving. Still, many business problems can be solved, using both the technologies. Arguably, exposing students concurrently to spreadsheet and database solutions may contribute to quite effective learning reinforcement. This paper shows how to solve a classic payroll-computing problem in a spreadsheet and in database systems. It is inspired by successful classroom experience of problem solving done with the two technologies. Solution examples, developed in Microsoft Excel, Access and in MySQL are presented and discussed.

Keywords: Problem solving, business case, spreadsheet, database, business curriculum.

LEARNING OBJECTIVES

Problem solving is one of the most desirable skills expected from business graduates. For example, at Western New England University, in an AACSB accredited Business program, the number one and five objectives are "To solve business problems by thinking critically and applying principles of effective decision making" and "To apply information technology concepts and tools to support business problem-solving and decision-making", respectively. One of the first courses to address this objective is BIS 102, "Problem Solving with Business Tools". There are three areas in which this course addresses business problem solving: (1) spreadsheet modeling, (2) database design and query development, and (3) business analytics. The latter is explored using both the spreadsheet and database technologies.

The learning case, presented in this paper, deals with a Human Resource (HR) problem of computing gross salaries for hourly paid employees. Upon completion of this case, student are expected to learn how to:

- model payroll data,
- integrate the data,
- compute weekly salaries.

These objectives can be accomplished using spreadsheet or database tools. Typically, the contemporary textbooks, for example, (Adamski, 2011), (Gaskin, 2014), (Parsons, 2011), (Poatsy, 2013), develop expertise for solving this problem in both a spreadsheet and in a database. However the solution implementations are presented independently. Arguably, presenting the implementations concurrently, side-by-side, is expected to contribute to better understanding of the problem and its solution as well as to improved skills in utilizing both the spreadsheet and database systems.

CASE PAYROLL - PROBLEM STATEMENT

A small business company, Sunny Energy, Inc. employs a few managers, assistants, technical specialists, and production workers. The company has been handling its payroll in a spreadsheet in a sort of primitive way, involving lots of manual operations. With the recent expansion, the payroll assistant, Alice Ray, realized that the current system became very difficult to manage, especially with respect to the production workers who get paid based on their hours worked with overtime options. Salaries of other employees are relatively stable and unchallenging. Alice had an idea to contact her former professor, Dr. Baran, who she remembered from her Microcomputer Application class. Her boss, Mr. Bear, approved her idea and she met with Dr. Baran. To make this story short, Dr. Baran recommended his best student, Nick Bright to assist Alice with her problem solving. He only had one advice for Nick and Alice. He said that they must develop more than one solution and for the most of it the solutions should be data integrity friendly. Nick knew right away what his professor meant. He was sure that one of the solutions would have to involve a database, for example Microsoft Access or MySQL. He mentioned it to Alice, and she replied that she was not familiar with any database technology and, while she was willing to consider it, she asked if it were possible to first develop a working prototype in Excel.

DATA MODELING

Whether spreadsheet or database is to be applied, one needs to specify the problem in a more concise language. Wrapping it all up, the problem can be described, using simple SPO[1] type statements:

- Production Worker *belongs to* a Department.
- Production Worker *works* Hours.
- Production Worker *gets* Weekly Pay.

Based on this high level design, one can identify a few related entities: Production Worker (further referred to as `Worker`), `Department`, Hours (further referred to as `TimeSheet`), and `WeeklyPay`.

In a prototype solution, the entities (`Department`, `Worker`, `TimeSheet`, and `WeeklyPay`) are described with minimal sets of their attributes:

- `Department (d_id, title).`
- `Worker (w_id, d_id, firstName, lastName, hourlyPayRate).`
- `TimeSheet (ts_id, w_id, workDate, hoursWorked).`
- `WeeklyPay (wp_id, w_id, payYear, weekNumber, grossPay).`

Other relevant details, used to manage these human resources can be easily added later.

A visual representation of the above design, can be shown, using an Entity Relationship (ER) model or a Unified Modeling Language (UML) model. (Teorey at al., 2006, p. 9). MySQL Workbench is a free but powerful tool that can be used to develop such diagrams (in both the ER and UML format). Figure 1 shows an initial diagram, containing unrelated entities. Attributes `Worker.d_id`, `TimeSheet.w_id`, and `WeeklyPay.wp_id` are not shown [explicitly] since they are derived from the relationships between the entities. Figure 2 shows a complete diagram, including entity relationships. A detail instruction about developing such diagrams is provided in (Letkowski, 2015 A). Unfortunately, Access has limited functionality for designing databases which is why the Payroll ER model is developed in MySQL Workbench (Letkowski, 2015 B).

SPREADSHEET IMPLEMENTATION

Doing tables in Excel to process data entities is a straight forward task. One takes the attribute names of an entity to create a table header and all detail data is entered in the attribute columns below the header. Figure 3 shows two tables: `Department` and `Worker`. It is always a

[1] SPO stands for Subject – Predicate – Object. RDF and its extension OWL are a formal implementations of such a language. Interesting applications of these languages are presented in (Matheus, at al., 2005 and 2006).

good idea to name tables in Excel. The tables are named in two ways, as `Department`, `Worker` and `_Department`, `_Worker`, respectively. The former names refer to the whole tables, including the headers, and the latter—only to the data rows. Depending on circumstances one can use conveniently one name or the other. For example, while importing the tables to Access, one would use the names referencing the entire tables (e.g. `Department` and `Worker`). When applying Excel functions such as `VLookup()`, `Index()`, etc., one would use the names only referencing the data rows (e.g. `_Department` and `_Worker`).

Figures 4 and 5 show fragments of an extended `TimeSheet` table. This table includes one extra column (`weekNumber & w_id`), combining the week number (`weekNumber`) and worker ID (`w_id`). The week number values are derived from the `workDate` values. This extra column contains unique identifiers of the workers' week records that will be used to retrieve work hours for particular weeks and worker IDs. For example, a combination of week number 2 and worker ID 1 identifies all workdays of this week for this worker. The sum of all hours worked, associated with `weekNumber & w_id` = "`2-1`", represents the total number of hours worked by worker 1 during the week 2 of the year. This extra column is built, using formula `=WEEKNUM(C2)&"-"&B2`, being replicated throughout the entire column. Following the naming pattern applied to the first two tables, this table is named as `TimeSheet` and `_TimeSheet`, respectively. When importing this table to Access the column `weekNumber & w_id` should be ignored.

The last table, `WeeklyPay`, is shown in Figure 6. The first column contains serial numbers (IDs) of the Payroll transactions. The second column is a dynamic copy of the worker ID column defined in the `Worker` table. The table also includes two intermediate columns (`hourlyPayRate` and `weekHours`) that facilitate calculations of the gross pay values (in column `grossPay`). The first of these two columns retrieves the hourly rate of pay from table `_Worker`, using the workers ID (w_id). It is done by means of the following function: `=VLookup(B5,_Worker,5)`[2]. Notice that 5 stands for the column number of the `_Worker` table from which the hourly rate of pay is being retrieved. Recall that name `_Worker` refers just to the data rows of the source table stored on sheet `Worker`. The second column, `weekHours`, shows calculated weekly hours for the given week number (in cell `E3`) and worker ID (in cell `B5`). It uses a conditional function, SumIf()[3], whose first argument refers to column `weekNumber & w_id` of the `TimeSheet` table to be matched by the second argument, combining the week number and worker ID, followed by third argument, referencing the work

[2] The VLookup() function is known for stirring fearful emotions among students even so it is extensively covered by the popular spreadsheet textbooks (Gaskin, at al., 2014, p. 251-255), (Poatsy-Mulbery, at al., 2014, p. 184-186), (Parsons, at al., 2011, p. EX388-394) . The 2013 edition of Excel offers an alternative to this function by means of Relationships but it can only be explored within Pivot Tables.

[3] The SumIf () function is also quite challenging. It takes a good visualization for the students to perfectly understand how this function works. It is well documented and explained in the spreadsheet textbook mentioned above, (Gaskin, at al., 2014, p. 481-482), (Poatsy-Mulbery, at al., 2014, p. 467-468), (Parsons, at al., 2011, p. EX407-408).

hours to be totaled:

```
=SUMIF(TimeSheet!$E$2:$E$2593,"="&$E$3&"-"&B5,TimeSheet!$D$2:$D$603).
```

Notice that the way the second argument is constructed, is identical to the way the values in the `TimeSheet!$E$2:$E$2593` range (the first argument) are defined. This is absolutely necessary in order to ensure consistency of the references and ultimately—the correctness of calculations. The size of the third argument, `TimeSheet!$D$2:$D$603`, is expected to be the same as that of the first one.

The last column in the `WeeklyPay` table, `grossPay`, fulfills the main objective of this whole application. It shows calculated gross weekly pay amounts. The formula used in this column is based on the following payroll rule, taking into account the base pay and two types of the overtime pay, if any:

$$p = \begin{cases} r \cdot h & \text{if} \quad h \le 40 \\ r \cdot h + r \cdot (h-40)/2 & \text{if} \quad 40 < \mathrm{h} \le 60 \\ r \cdot h + r \cdot 10 + r \cdot (h-60) & \text{if} \quad \mathrm{h} > 60 \end{cases} \qquad \text{(i)}$$

In order to simplify this rule, variable `r` and `h` are mapped from `hourlyPayRate` (defined in table `Worker`) and `hoursWorked` (defined in table `TimeSheet`), respectively. In short, the weekly gross pay is a straight product of the hourly pay rate and the number of hours worked for those who worked during the whole week no more than `40` hours. For the hours above `40` and up to `60`, the workers get paid extra at 50% of the rate (`r/2`). Finally, those who worked more than `60` hours during the week get extra pay for `20` hours between `40` and `60` at the 50% of the rate (`r·20/2 = r·10`) plus additional pay for hours above `60` at the 100% of the rate, `r·(h-60)`. Taking advantage of the Excel's `Max()` function, the above formula can be implemented as:

```
=ROUND(C5*(D5+(MAX(D5,40)+MAX(D5,60)-100)/2),2)
```

This formula is replicated (copied/pasted) in the whole `grossPay` column. Interestingly, without the two extra columns, `hourlyPayRate` and `weekHours`, this formula would become quite monstrous:

```
=ROUND(VLOOKUP(B5,_Worker,5)*
(SUMIF(TimeSheet!$E$2:$E$2593,"="&$E$3&"-"&B5,
TimeSheet!$D$2:$D$603)+(MAX(SUMIF(TimeSheet!$E$2:$E$2593,"="&$E$3&"-
"&B5, TimeSheet!$D$2:$D$603),40)+MAX(SUMIF(TimeSheet!$E$2:$E$2593,
"="&$E$3&"-"&B5,TimeSheet!$D$2:$D$603),60)-100)/2),2)
```

It is customary to use intermediate cells in a spreadsheet application in order to simplify formulas and make them more understandable and maintainable. This approach fall into a framework of solving problems via a systematic decomposition.

The spreadsheet solution, presented so far, is quite generic. It can be employed in programs other than Excel (Google Sheets, Apache OpenOffice, etc.). Although the main objective (to compute the weekly gross payroll) has been accomplished a few unresolved issues must be addressed and resolved before this solution can be applied. For example, one could ask: what will happen next week? The dates (Beginning Date and End Date) will be changed, causing the week number to change as well. The weekly payroll for the previous week will vanish, while being overwritten by the new week's results. A simple solution to this problem would be to create a new worksheet that will accumulate sequentially all weekly outcomes processed so far. However, an in-depth coverage of this and similar payroll management issues goes beyond the scope of this paper. Needless to say, a database solution may shed some extra light at these payroll maintenance problems.

DATABASE IMPLEMENTATION

The first step in structuring a database solution is to create a database and add raw data. Since the Excel tables, explored so far, already contain the raw data, it make sense to import the data from the Excel workbook to an Access or MySQL database[4]. One has to be extra careful when getting Excel tables that contain calculated columns. Such columns should be discarded. The Access's import and MySQL add-in's export wizards provide an easy way to do it. Figure 7 shows fragments of the three tables (`Department`, `Worker`, and `TimeSheet`) imported from the Excel workbook into an Access database. Notice that the first two tables were imported entirely and the third one—partially. The `weekNumber & w_id` column of the `TimeSheet` table, as a calculated column, was omitted. The database schema diagram is shown in Figure 8.

Getting even basic data for a weekly payroll of a year is a quite complex task. It makes sense to break the problem into smaller tasks, then complete that tasks and aggregate them into the final solution. Such a problem solving process is shown below for both Access and MySQL.

Access Solution

The first task is to get the right for the hours worked by each worker in the selected week of the year. Figure 9 shows a query, fetching worker IDs and hours work for the selected period. The SQL version of this query is as follows:

```
SELECT w_id, hoursWorked AS h
FROM TimeSheet
WHERE (((DatePart("ww",[workDate]))=[Week Number]) AND
((Year([workDate]))=[Payroll Year]));
```

Notice that function `DatePart("ww",[workDate])` is used in this query in order to extract the week number from each date. In Excel, function `=WEEKNUM()` was used. Access is not equipped with such a function. Also notice that this query utilizes two parameters, `[Week Number]` and `[Payroll Year]`. Values of these parameters are entered at the query's run time. This query is saved as view `WorkersAndHoursWnAndY` (workers and hours for week number and year).

[4] Importing data from Excel into Access is a pretty straight-forward task. Many common textbooks provide excellent set of examples (Adamski, at al., 2011, p. AC76-79), (Poatsy-Krebs, at al., 2014, p. 157-160). The latest MySQL setup program can also install the MySQL for Excel Database add-in that can be used to import data from Excel to MySQL databases (MySQL, 2015).

The output of the first query (Figure 10) is used as input for the second query, whose purpose is to aggregate (subtotal) hours worked for each of the workers (Figure 11). The SQL statement for this query is as follows:

```
SELECT w_id, Sum(h) AS totalHours
FROM WorkersAndHoursWnAndY
GROUP BY w_id;
```

This is a classic subtotal query. It is saved as view `WorkersAndTotalHoursWnAndY` (workers and total hours for week number and year). Its output is shown in Figure 12.

Going with baby steps, the job of the third query is to collect all the data needed for the final weekly pay report, except of the `grossPay` column (Figure 13):

```
SELECT Department.d_id, Department.name, Worker.w_id, Worker.firstName,
  Worker.lastName, Worker.hourlyPayRate AS r,
  WorkersAndTotalHoursWnAndY.totalHours AS h
FROM Department INNER JOIN
  (Worker INNER JOIN WorkersAndTotalHoursWnAndY ON Worker.w_id =
  WorkersAndTotalHoursWnAndY.w_id)
ON Department.d_id = Worker.d_id;
```

This is a typical `INNER JOIN` query, combining related data from two tables, `Department`, `Worker`, and one view, `WorkersAndTotalHoursWnAndY`. The name of this query (view) is `PDWY`. Figure 14 shows its output.

The touch-down for the run to the Weekly Gross Payroll solution is represented by the last query, `PayrollForWnY` (Figure 15). It adds just one calculated column that computes the weekly gross pay based on the rule (i), taking into account the base pay and the over-time pay, if any:

```
SELECT PDWY.d_id, PDWY.name, PDWY.w_id, PDWY.firstName, PDWY.lastName,
  PDWY.r AS hourlyRate, PDWY.h AS weeklyHours,
  Round([r]*([h]+(IIf([h]>40,[h],40)+IIf([h]>60,[h],60)-100)/2),2)
    AS grossPay FROM PDWY
ORDER BY PDWY.w_id;
```

The `grossPay` column is defined in a similar way to the Excel's corresponding column. There is one important difference thou. One may not use the `Max()` function in this context as Access and other database systems use `Max()` as an aggregate function. Instead an `IIF()` function is used. It is equivalent to Excel's `IF()` function. In this case, it picks up the greater of two values. The output of this query (Figure 16) makes up the basis for the Weekly Payroll report.

Looking at this solution, Alice realized that it is actually not that difficult. She would probably quickly learn how to live with it. Nick uttered that a MYSQL solution is also interesting.

MySQL Solution

MySQL implements parametric queries either through global variables or, more commonly, via stored procedures. The following SQL code shows how to gradually develop a stored procedure that will produce the same output as the last Access query (`PayrollForWnY`).

First, prototype views (for specific dates), solving sub-problems, are developed, next all the views are aggregated into a single query which, finally, is used to develop the stored procedure.

In MySQL, the query language is closer to the SQL standard and, in many aspects, it is more flexible and powerful than Access query tools. In particular, it is easier to produce queries for grouped data (e.g. subtotals). The following is the first query (view) that returns worker IDs along with their hourly pay rates and total hours between tow dates. Nick decided to show a different, more flexible, approach to handling dates. Instead of using year and week number, he chose two consecutive dates. This way, it will be easy to produce a Payroll report for any period. The first query/view, `WHR`, uses the dates, `'2015-03-23'` and `'2015-03-29'`, that are equivalent to week number 13 and year 2015.

```
-- Query 1: WHR
CREATE OR REPLACE VIEW WHR AS
SELECT w_id, hourlyPayRate AS r, Sum(hoursWorked) AS h
FROM Worker INNER JOIN TimeSheet USING(w_id)
WHERE workDate BETWEEN '2015-03-23' AND '2015-03-29'
GROUP BY w_id;
SELECT * FROM WHR;
```

By the way, the same output would be produced with the `WHERE` clause, using the week number and year:

```
WHERE WEEK(workDate,3)=13 AND YEAR(workDate)=2015
```

Notice that the second argument of the `WEEK()` function defines mode 3 which sets the first day of the week to Monday and starts week numbers with 1.

The second view, `WHRP`, adds the `grossPay` column and it uses the first view. Notice that the way the periodic gross pay is calculated is similar to that in Access. MySQL uses the `IF()` function (instead of the `IIF()` one):

```
-- Query 2: WHRP
CREATE OR REPLACE VIEW WHRP AS
SELECT w_id, r, h,
  Round(r*(h+(If(h>40,h,40)+If(h>60,h,60)-100)/2),2) AS grossPay
FROM WHR;
SELECT * FROM WHRP;
```

The third and last query includes other details (departmental and personal). This is a typical `INNER JOIN` query:

```
-- Query 3: DWHRP
CREATE OR REPLACE VIEW DWHRP AS
SELECT Department.d_id, Department.name, Worker.w_id, Worker.firstName,
  Worker.lastName, WHRP.r AS hourlyRate, WHRP.h AS hoursWorked, WHRP.grossPay
FROM Department INNER JOIN Worker USING(d_id) INNER JOIN WHRP USING(w_id)
ORDER BY Worker.w_id;
SELECT * FROM DWHRP;
```

The process of aggregating the views is simple in principle but complicated in execution. One replaces the names of the views with SQL code (`SELECT` statements) that define the views. SQL views are referred to virtual tables. After the view name is replaced with its `SELECT` statement, the latter becomes a sub-query. It is important to remember that each subquery must be enclosed in parentheses and be assigned an alias. Thus, in the second query, the `FROM` clause it rewritten, replacing name `WHR` with the query stranding behind it, resulting in:

```
SELECT w_id, r, h,
  Round(r*(h+(If(h>40,h,40)+If(h>60,h,60)-100)/2),2) AS grossPay
FROM (SELECT w_id, hourlyPayRate AS r, Sum(hoursWorked) AS h
FROM Worker INNER JOIN TimeSheet USING(w_id) WHERE workDate BETWEEN '2015-03-
23' AND '2015-03-29' GROUP BY w_id) WHR;
```

Using the above query as replacement of view name `WHRP` in Query 3 (`DWHRP`) yields this query:

```
SELECT Department.d_id, Department.name, Worker.w_id, Worker.firstName,
 Worker.lastName, WHRP.r AS hourlyRate, WHRP.h AS hoursWorked, WHRP.grossPay
FROM Department INNER JOIN Worker USING(d_id)
 INNER JOIN (SELECT w_id, r, h,
  Round(r*(h+(If(h>40,h,40)+If(h>60,h,60)-100)/2),2) AS grossPay
  FROM (SELECT w_id, hourlyPayRate AS r, Sum(hoursWorked) AS h
   FROM Worker INNER JOIN TimeSheet USING(w_id)
   WHERE workDate BETWEEN '2015-03-23' AND '2015-03-29'
   GROUP BY w_id) WHR) AS WHRP
  USING(w_id)
ORDER BY Worker.w_id;
```

The final step is to inject this complicated query into a stored procedure. This will enable replacement of the specific dates with parameters (`p_startDate` and `p_endDate`):

```
-- Stored Procedure:
DROP PROCEDURE IF EXISTS WeeklyPayroll;
DELIMITER |
CREATE PROCEDURE WeeklyPayroll( IN p_startDate DATETIME, IN p_endDate
DATETIME)
BEGIN
  SELECT Department.d_id, Department.name, Worker.w_id, Worker.firstName,
   Worker.lastName, WHRP.r AS hourlyRate, WHRP.h AS hoursWorked, WHRP.grossPay
  FROM Department INNER JOIN Worker USING(d_id)
   INNER JOIN (SELECT w_id, r, h,
    Round(r*(h+(If(h>40,h,40)+If(h>60,h,60)-100)/2),2) AS grossPay
    FROM (SELECT w_id, hourlyPayRate AS r, Sum(hoursWorked) AS h
     FROM Worker INNER JOIN TimeSheet USING(w_id)
     WHERE workDate BETWEEN p_startDate AND p_endDate
     GROUP BY w_id) WHR) AS WHRP
    USING(w_id)
  ORDER BY Worker.w_id;
END
|
DELIMITER ;
CALL WeeklyPayroll('2015-03-23', '2015-03-29');
```

This procedure can be executed in any MySQL client-console or by an application program written in Java, PHP, etc.

Now, having three different solutions to the payroll problem it is a decision time to choose the most efficient and convenient one. It is not an easy decision. One has to take all technological and organizational expectations, restrictions and requirements.

CONCLUSION

Allice took time to review and analyze both the spreadsheet and database solutions. While she noticed that the solutions represent significant improvement compared to her current system, she also realized that more work is needed in order to further improve her payroll generating process. She looked at additional requirements of the payroll system and, after exhaustive discussion with Nick, she concluded that it would be not too difficult to satisfy them by extending the current spreadsheet and/or database tables. At this point, Alice would prefer to stay with the spreadsheet solution. After all, she knows well how to deal with Excel. In spite of Nick's recommendation, she was not ready for a database solution. Still, there was one peculiar piece of her puzzle that bothered her and she did know how to handle. There seemed to be not an easy fix for managing updates of the hourly pay rates of the workers. She arranged for another meeting with Nick in order to find out how to handle this predicament.

During the meeting, after a few minutes of silence, only interrupted by strange noises coming for Nick's brain, Nick realized that they have made a serious design error. This issue of dynamic pay rate should have been resolved when they were structuring the logical data model. Obviously, the hourly rate of pay should be considered to be an entity on its own rather than a simple property of entity Worker. For every instance of Worker there should be at least one instance of the pay rate. When a worker gets a raise, the new rate should be recorded but the old ones should not be forgotten. Taking this requirement into account, the old ER diagram was modified as shown in Figure 17. Nick quickly updated the database in MySQL. The data changes are shown in Figure 18.

Using a similar process of decomposition and aggregation, Alice and Nick developed the following SQL code:

```
-- Query 1: EffectiveDate
CREATE OR REPLACE VIEW EffectiveDate AS
SELECT w_id, MAX(effDate) as theDate
FROM PayRate
WHERE effDate <= '2015-03-23'
GROUP BY w_id;
SELECT * FROM EffectiveDate;
```

This query determines the dates of the most recent hourly rate updates for each worker that were made before or on the given date (here `'2015-03-23'`).

```
-- Query 2: EffectiveRate
CREATE OR REPLACE VIEW EffectiveRate AS
SELECT w_id, r
FROM PayRate JOIN EffectiveDate Using(w_id)
WHERE effDate = theDate
ORDER BY w_id;
SELECT * FROM EffectiveRate;
```

Query 2 pulls the up-to-date hourly rates for each worker.

```
-- Query 3: EffectiveHours
CREATE OR REPLACE VIEW EffectiveHours AS
SELECT w_id, Sum(hoursWorked) AS h
FROM TimeSheet
```

```
WHERE workDate BETWEEN '2015-03-23' AND '2015-03-29'
GROUP BY w_id;
SELECT * FROM EffectiveHours;
```

This query gets the total hours worked for each worker between the given dates.

```
-- Query 4: EffectiveRateHours
CREATE OR REPLACE VIEW EffectiveRateHours AS
SELECT w_id, r, h
FROM EffectiveRate JOIN EffectiveHours USING (w_id);
SELECT * FROM EffectiveRateHours;
```

Query 4 merges the outcomes of queries `EffectiveRate` and `EffectiveHours`, providing intermediate result similar to that of view `WHRP`.

```
-- Query 5: EffectivePayroll
CREATE OR REPLACE VIEW EffectivePayroll AS
SELECT d_id, name, w_id, firstName, lastName, r as hourlyRate, h as
totalHours, Round(r*(h+(If(h>40,h,40)+If(h>60,h,60)-100)/2),2) AS grossPay
FROM Department JOIN Worker USING (d_id) JOIN EffectiveRateHours USING
(w_id);
SELECT * FROM EffectivePayroll;
```

This final query/view does the similar job as the previous query `DWHRP`. As shown in the previous section, the queries/view can be aggregated into just one big query embedded in a stored procedure, for example:

```
-- Stored Procedure WeeklyPayroll:
DROP PROCEDURE IF EXISTS WeeklyPayroll;
DELIMITER |
CREATE PROCEDURE WeeklyPayroll( IN p_startDate DATETIME,
                                IN p_endDate DATETIME)
BEGIN
  SELECT d_id, name, w_id, firstName, lastName, ERH.r as hourlyRate,
    ERH.h as totalHours,
    Round(r*(h+(If(h>40,h,40)+If(h>60,h,60)-100)/2),2) AS grossPay
  FROM Department JOIN Worker USING (d_id) JOIN
    (SELECT w_id, r, h
     FROM (SELECT w_id, r
       FROM PayRate JOIN
         (SELECT w_id, MAX(effDate) as theDate
          FROM PayRate
          WHERE effDate <= p_startDate
          GROUP BY w_id) AS ED
          USING(w_id)
       WHERE effDate = theDate) AS ER JOIN
       (SELECT w_id, Sum(hoursWorked) AS h
       FROM TimeSheet
       WHERE workDate BETWEEN p_startDate AND p_endDate
       GROUP BY w_id) AS EH
       USING (w_id)) AS ERH
     USING (w_id)
  ORDER BY w_id;
END
|
DELIMITER ;
CALL WeeklyPayroll('2015-03-23', '2015-03-29');
```

Figure 19 shows the store procedure's output.

Similar modifications can be done to the Excel and Access applications. More challenging would be getting the Excel application to account for the dynamic pay rates. These challenges are left to be considered as exercises.

In summary, before accepting a software solution, one should explore different options and choose the one that satisfies the organizational goals as well as human and technological constraints. Off-the-shelf software alternatives should also be considered.

Regarding students' learning issues, a situation experienced by the author may be of interest. In the course, mentioned in the first section, the student first attempted to learn Access, including fundamentals of database design, development and query applications. Initially, they showed not too much enthusiasm and many of them could not wait to jump onto Excel. The initial chapters of Excel went smoothly and all the students were quite happy. Significant difficulties surfaced when exploring spreadsheet lists, tables, aggregate and conditional functions, what-if analysis and pivot tables. However, it was not until the final exam review. The students were exposed to problems like the one presented in this paper, when a miracle happened. They were given side-by-side examples of problem solutions in Excel and Access. Kind of surprising, they started showing more enthusiasm towards Access. Moreover, when taking an optional, final-exam-exemption test, on average, the students received better grades for the Access part than for the Excel part.

REFERENCES

Adamski, J. J., Finnegan. K. T. (2011). *New Perspectives on Microsoft Access 2010, Brief.* Boston: Course Technology - Cengage Learning.

Gaskin, S., Vargas, A., Geoghan, D. (2014). GO! With Microsoft® Excel 2013 – Comprehensive. Boston: Pearson Education, Inc. as Prentice Hall.

Letkowski, J. (2015 A). Doing database design with MySQL. Journal of Technology Research. ISSN Online: 1941-3416 (http://www.aabri.com/jtr.html, http://www.aabri.com/manuscripts/142002.pdf) Volume 6.

Letkowski, J. (2015 B). Challenges in database design with Microsoft Access. Academic and Business Research Institute Conference, Orlando 2015, January 1-3 (http://www.aabri.com/OCProceed2015.html, OC2015ES, http://www.aabri.com/OC2015Manuscripts/OC15065.pdf).

Letkowski, J. (2015 C) Spreadsheet and database resources for this paper. Retrieved from: http://letkowski.us/pub/conference/aabri/2015/savannah/

Matheus C., Baclawski K., Kokar M., Letkowski J. (2005) Using SWRL and OWL to Capture Domain Knowledge for a Situation Awareness Application Applied to a Supply Logistics Scenario. In Rules and Rule Markup Languages for the Semantic Web First International Conference, A. Adi, S. Stoutenburg (Ed), pages 130-144. Lecture Notes in Computer Science 3791:130-144. Springer-Verlag. (November 10-12, 2005).

Matheus C., Dionne B., Parent D., Baclawski K., Kokar, M. (2006) BaseVISor: A Forward-Chaining Inference Engine Optimized for RDF/OWL Triples. The 5th International Semantic Web Conference, Athens, GA, USA.

MySQL (2015). MySQL For Excel Database Add-in. Retrieved from: http://dev.mysql.com/doc/mysql-for-excel/en/index.html

Parsons, J. J., Oja, D., Ageloff, R., Carey, P. (2011). New Perspective on Microsoft® Excel® 2010. Boston: Course Technology, Cengage Learning.

Poatsy, M. N., Mulbery, K., Davidson, J., Lawson, R., Grauer, R.T. (2014). *Exploring: Microsoft Excel 2013, Comprehensive*. Boston: Pearson Education, Inc.

Poatsy, M. N., Krebs, C., Cameron, E., Williams, J., Grauer, R.T. (2014). *Exploring: Microsoft Access 2013, Comprehensive*. Boston: Pearson Education, Inc.

Teorey, T., Lightstone, S, Nadeau, T. (2006). Database Modeling & Design: Logical Design. San Francisco: Elsevier, Inc. as Morgan Kaufman Publishers.

Wikipedia-MySQLW (2014). MySQL Workbench. Retrieved from: https://en.wikipedia.org/wiki/MySQL_Workbench

APPENDIX

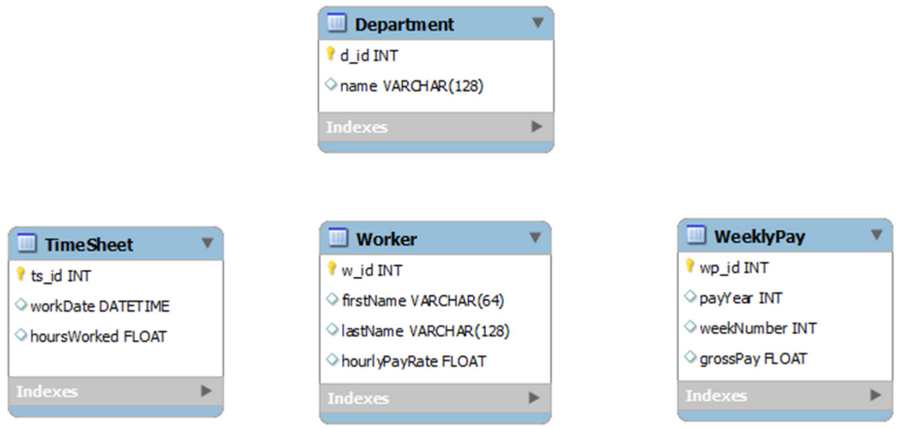


Figure 1. Data entities and their attributes.

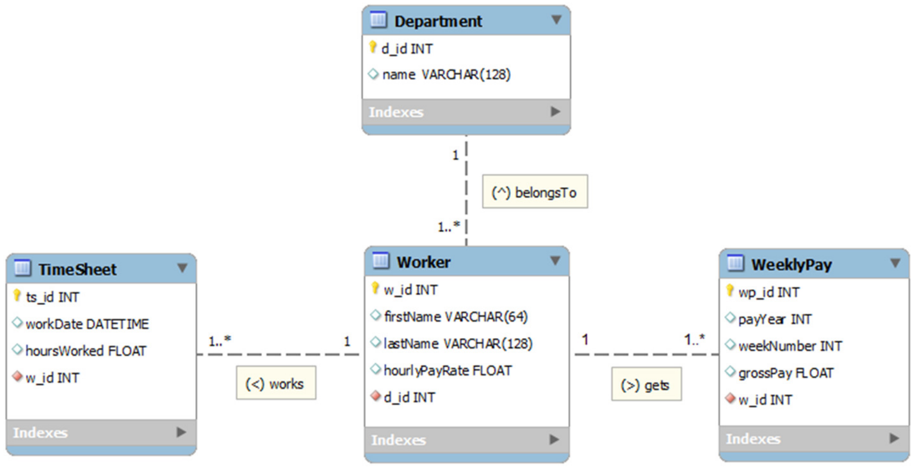
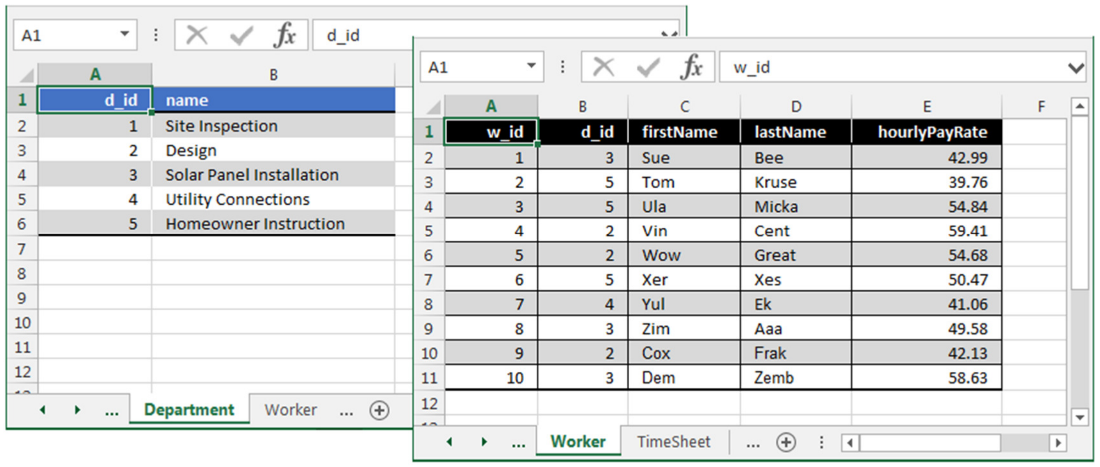


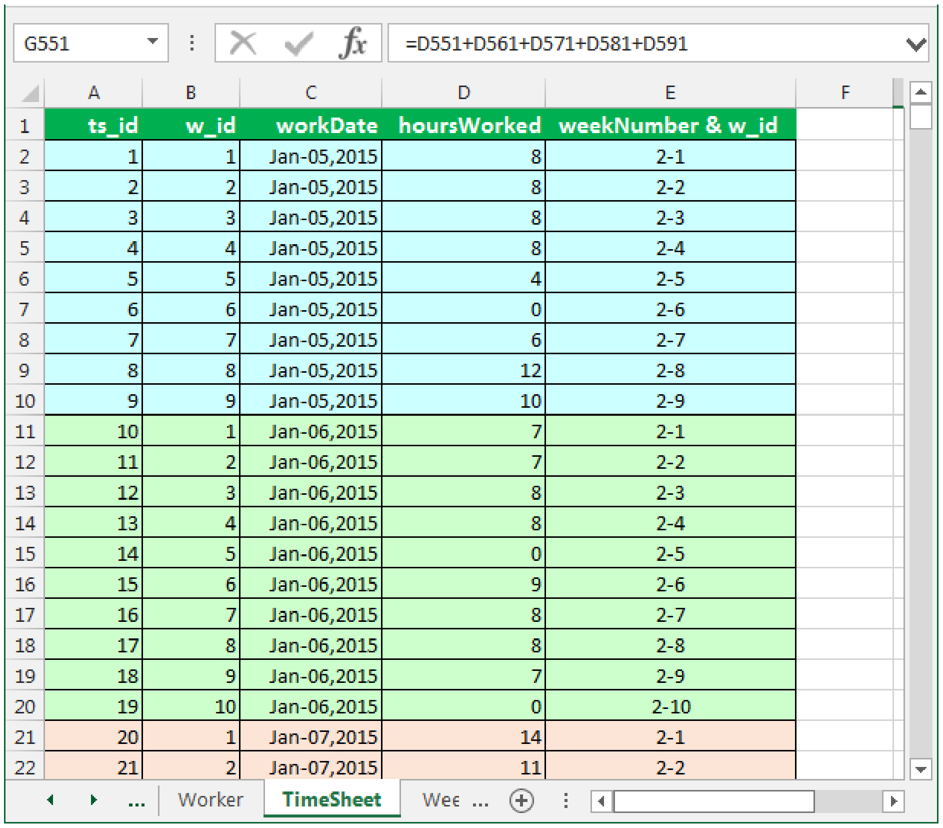
Figure 2. A complete ERD (Entity Relationship Diagram).



d_id	name
1	Site Inspection
2	Design
3	Solar Panel Installation
4	Utility Connections
5	Homeowner Instruction

w_id	d_id	firstName	lastName	hourlyPayRate
1	3	Sue	Bee	42.99
2	5	Tom	Kruse	39.76
3	5	Ula	Micka	54.84
4	2	Vin	Cent	59.41
5	2	Wow	Great	54.68
6	5	Xer	Xes	50.47
7	4	Yul	Ek	41.06
8	3	Zim	Aaa	49.58
9	2	Cox	Frak	42.13
10	3	Dem	Zemb	58.63

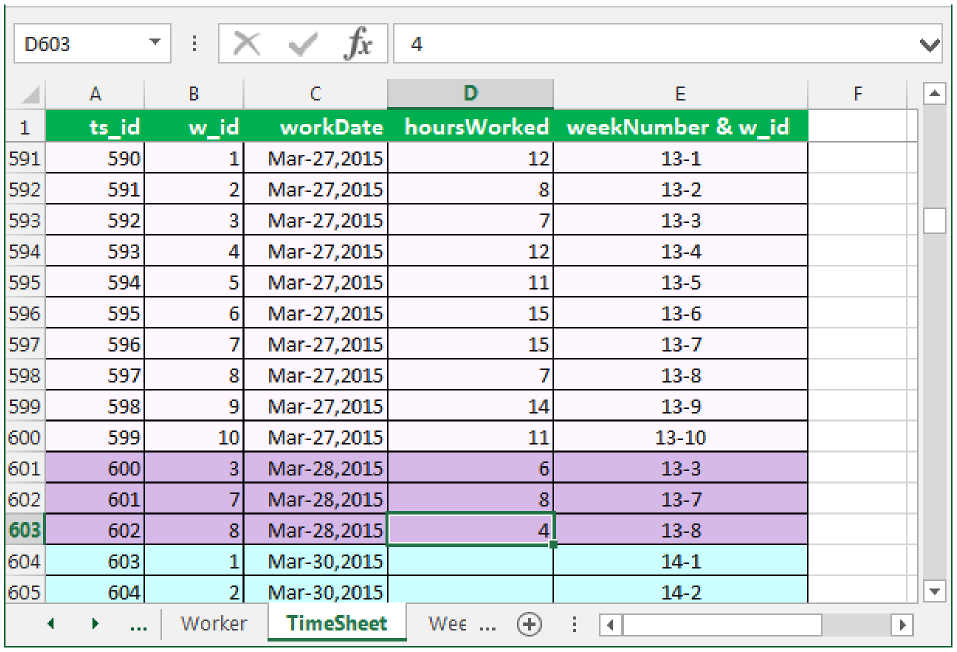
Figure 3 Entities `Department` and `Worker` implemented in Excel as free tables stored on two separate sheets.



G551 =D551+D561+D571+D581+D591

	A	B	C	D	E	F
1	ts_id	w_id	workDate	hoursWorked	weekNumber & w_id	
2	1	1	Jan-05,2015	8	2-1	
3	2	2	Jan-05,2015	8	2-2	
4	3	3	Jan-05,2015	8	2-3	
5	4	4	Jan-05,2015	8	2-4	
6	5	5	Jan-05,2015	4	2-5	
7	6	6	Jan-05,2015	0	2-6	
8	7	7	Jan-05,2015	6	2-7	
9	8	8	Jan-05,2015	12	2-8	
10	9	9	Jan-05,2015	10	2-9	
11	10	1	Jan-06,2015	7	2-1	
12	11	2	Jan-06,2015	7	2-2	
13	12	3	Jan-06,2015	8	2-3	
14	13	4	Jan-06,2015	8	2-4	
15	14	5	Jan-06,2015	0	2-5	
16	15	6	Jan-06,2015	9	2-6	
17	16	7	Jan-06,2015	8	2-7	
18	17	8	Jan-06,2015	8	2-8	
19	18	9	Jan-06,2015	7	2-9	
20	19	10	Jan-06,2015	0	2-10	
21	20	1	Jan-07,2015	14	2-1	
22	21	2	Jan-07,2015	11	2-2	

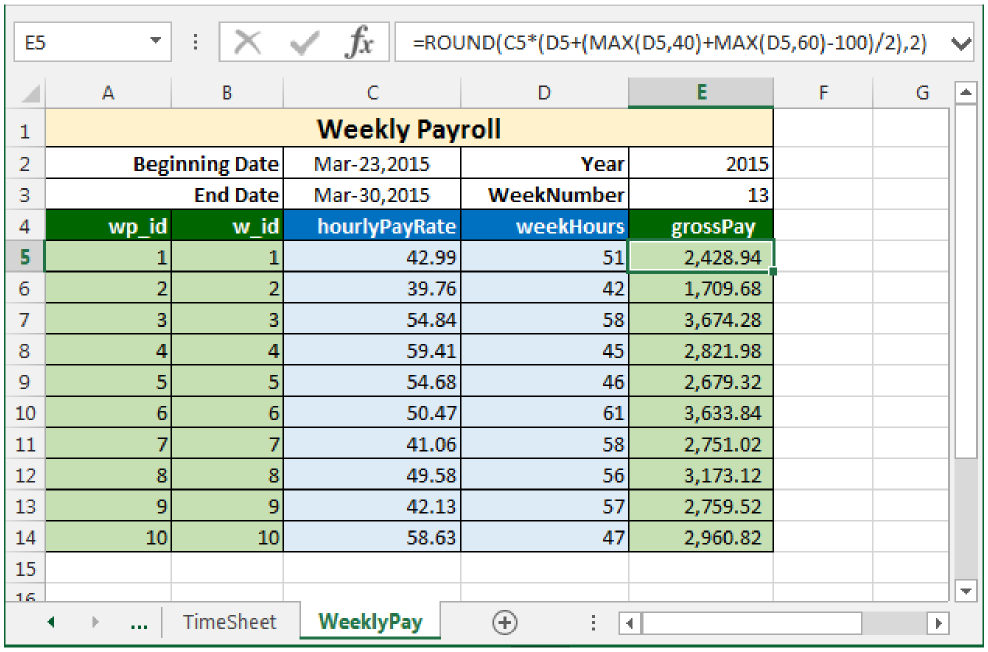
Figure 4. The top fragment of the `TimeSheet` table. Notice a new worker (`w_id=10`) hired on `Jan-06, 2015`.



D603 4

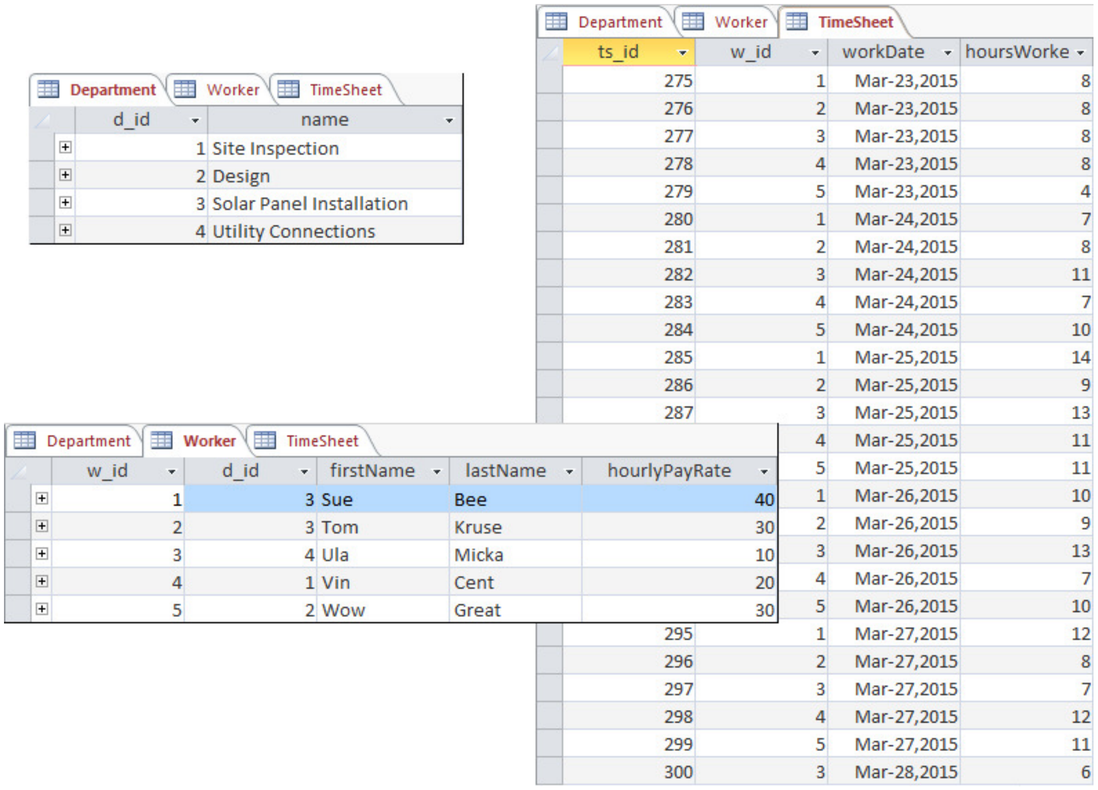
	A	B	C	D	E	F
1	ts_id	w_id	workDate	hoursWorked	weekNumber & w_id	
591	590	1	Mar-27,2015	12	13-1	
592	591	2	Mar-27,2015	8	13-2	
593	592	3	Mar-27,2015	7	13-3	
594	593	4	Mar-27,2015	12	13-4	
595	594	5	Mar-27,2015	11	13-5	
596	595	6	Mar-27,2015	15	13-6	
597	596	7	Mar-27,2015	15	13-7	
598	597	8	Mar-27,2015	7	13-8	
599	598	9	Mar-27,2015	14	13-9	
600	599	10	Mar-27,2015	11	13-10	
601	600	3	Mar-28,2015	6	13-3	
602	601	7	Mar-28,2015	8	13-7	
603	602	8	Mar-28,2015	4	13-8	
604	603	1	Mar-30,2015		14-1	
605	604	2	Mar-30,2015		14-2	

Figure 5. A fragment of the `TimeSheet` table with the `hoursWorked` data provided up to `March-28, 2015`. During the week number `13`, some of the workers also worked on `Saturday` (`March 28, 2015`).



Weekly Payroll				
	Beginning Date	Mar-23,2015	Year	2015
	End Date	Mar-30,2015	WeekNumber	13
wp_id	w_id	hourlyPayRate	weekHours	grossPay
1	1	42.99	51	2,428.94
2	2	39.76	42	1,709.68
3	3	54.84	58	3,674.28
4	4	59.41	45	2,821.98
5	5	54.68	46	2,679.32
6	6	50.47	61	3,633.84
7	7	41.06	58	2,751.02
8	8	49.58	56	3,173.12
9	9	42.13	57	2,759.52
10	10	58.63	47	2,960.82

Figure 6. Entity `WeeklyPay` implemented in Excel as free table. The two additional columns, `hourlyPayRate` and `weekHours` are incorporated here in order to simplify the formula for the `grossPay`.



d_id	name
1	Site Inspection
2	Design
3	Solar Panel Installation
4	Utility Connections

w_id	d_id	firstName	lastName	hourlyPayRate
1	3	Sue	Bee	40
2	3	Tom	Kruse	30
3	4	Ula	Micka	10
4	1	Vin	Cent	20
5	2	Wow	Great	30

ts_id	w_id	workDate	hoursWorke
275	1	Mar-23,2015	8
276	2	Mar-23,2015	8
277	3	Mar-23,2015	8
278	4	Mar-23,2015	8
279	5	Mar-23,2015	4
280	1	Mar-24,2015	7
281	2	Mar-24,2015	8
282	3	Mar-24,2015	11
283	4	Mar-24,2015	7
284	5	Mar-24,2015	10
285	1	Mar-25,2015	14
286	2	Mar-25,2015	9
287	3	Mar-25,2015	13
	4	Mar-25,2015	11
	5	Mar-25,2015	11
	1	Mar-26,2015	10
	2	Mar-26,2015	9
	3	Mar-26,2015	13
	4	Mar-26,2015	7
	5	Mar-26,2015	10
295	1	Mar-27,2015	12
296	2	Mar-27,2015	8
297	3	Mar-27,2015	7
298	4	Mar-27,2015	12
299	5	Mar-27,2015	11
300	3	Mar-28,2015	6

Figure 7. Access tables for the payroll application.

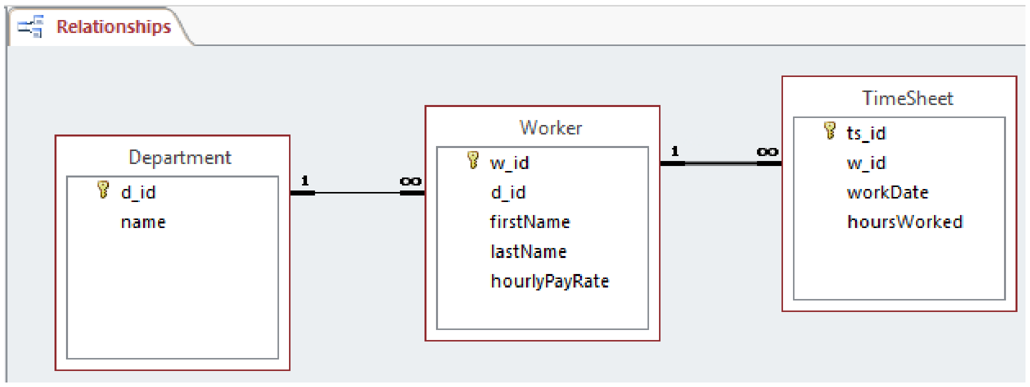
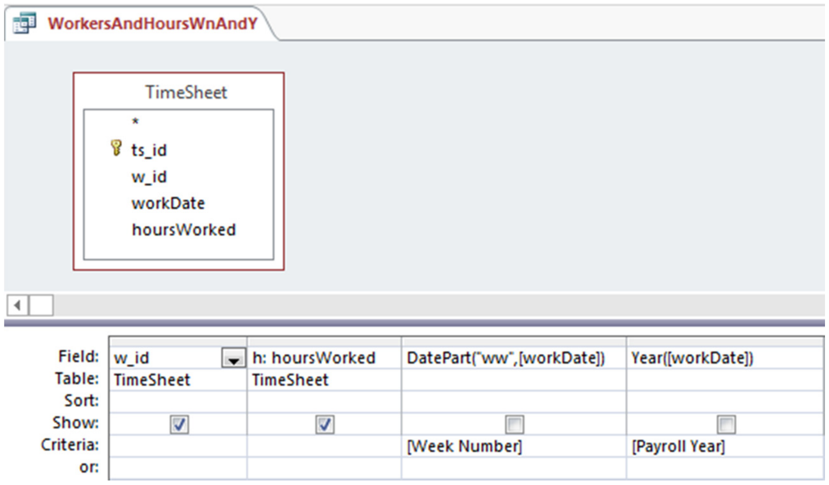
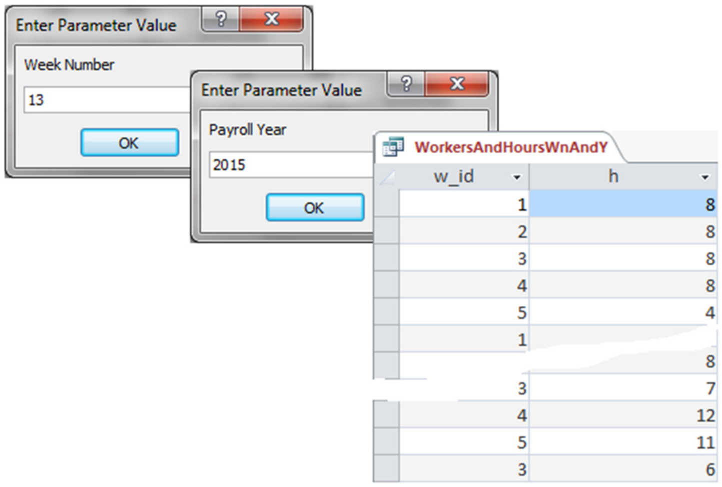


Figure 8. The database schema for the payroll application.



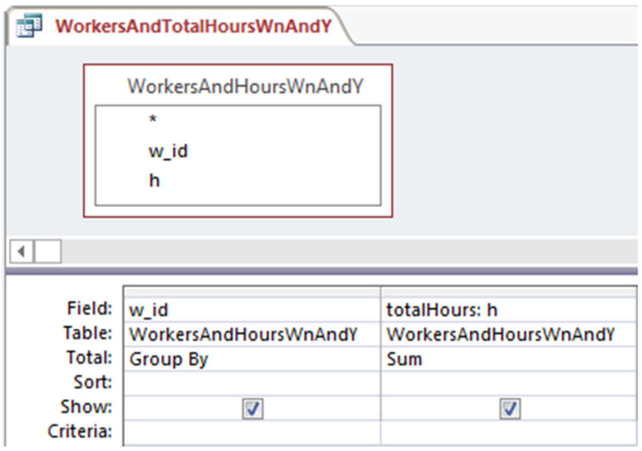
Field:	w_id	h: hoursWorked	DatePart("ww",[workDate])	Year([workDate])
Table:	TimeSheet	TimeSheet		
Sort:				
Show:	✓	✓	☐	☐
Criteria:			[Week Number]	[Payroll Year]
or:				

Figure 9. The first query, `WorkersAndHoursWnAndY`, providing the hours worked for all the workers within a given Year and Week Number. [Week Number] and [Payroll Year] are parameters of this query.



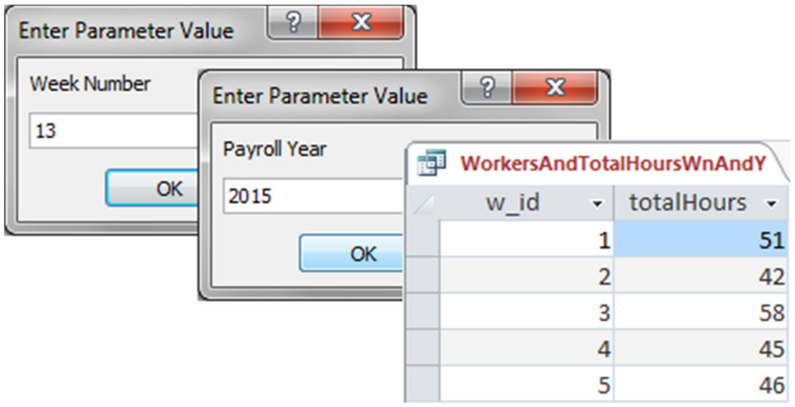
w_id	h
1	8
2	8
3	8
4	8
5	4
1	
	8
3	7
4	12
5	11
3	6

Figure 10. A fragment of the `WorkersAndHoursWnAndY` query's output.



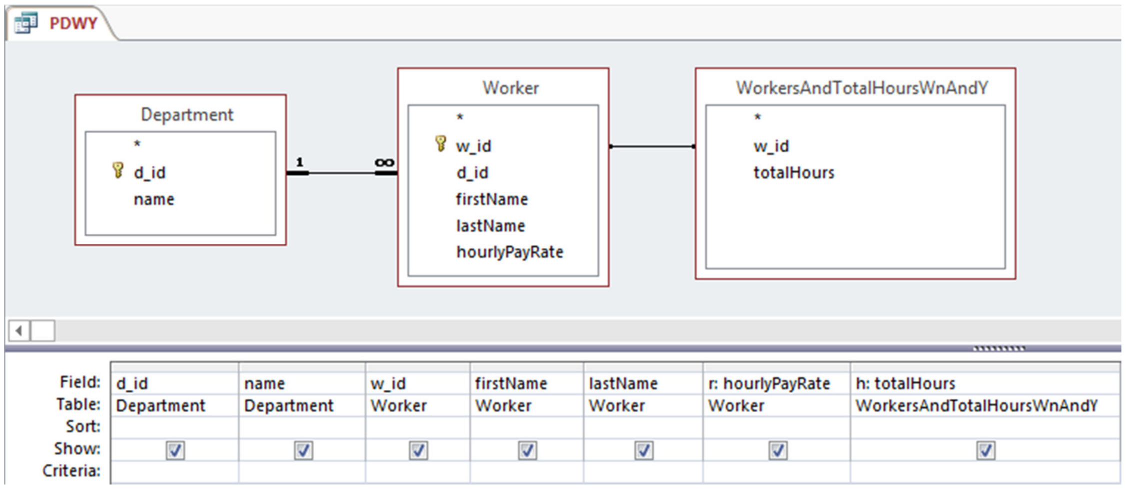
Field:	w_id	totalHours: h
Table:	WorkersAndHoursWnAndY	WorkersAndHoursWnAndY
Total:	Group By	Sum
Sort:		
Show:	✓	✓
Criteria:		

Figure 11. An aggregate query, `WorkersAndTotalHoursWnAndY`, to calculate hours worked subtotals for each of the workers.



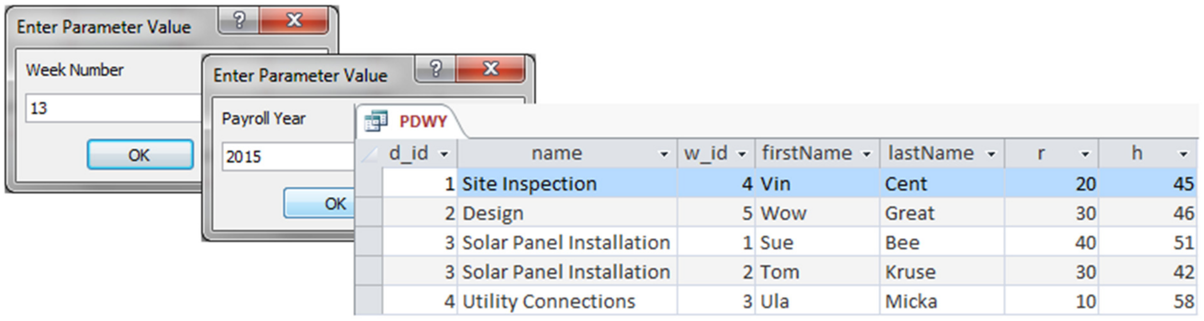
w_id	totalHours
1	51
2	42
3	58
4	45
5	46

Figure 12 The `WorkersAndTotalHoursWnAndY` query's output (the total hours worked by the workers 1, 2, …, 5, during week number 13 of year 2015.



Field:	d_id	name	w_id	firstName	lastName	r: hourlyPayRate	h: totalHours
Table:	Department	Department	Worker	Worker	Worker	Worker	WorkersAndTotalHoursWnAndY
Sort:							
Show:	✓	✓	✓	✓	✓	✓	✓
Criteria:							

Figure 13. A query, PWDY, to merge essential data necessary to generate the weekly payroll (only the gross pay column is missing).



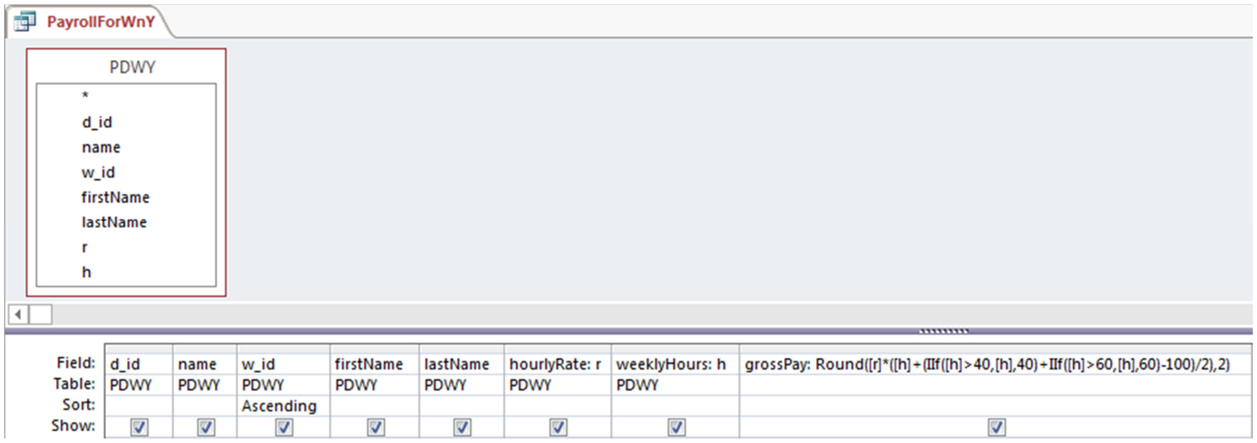
Enter Parameter Value — Week Number: 13 — OK

Enter Parameter Value — Payroll Year: 2015 — OK

PDWY

d_id	name	w_id	firstName	lastName	r	h
1	Site Inspection	4	Vin	Cent	20	45
2	Design	5	Wow	Great	30	46
3	Solar Panel Installation	1	Sue	Bee	40	51
3	Solar Panel Installation	2	Tom	Kruse	30	42
4	Utility Connections	3	Ula	Micka	10	58

Figure 14. The `PDWY` query's output. It show all columns necessary for the weekly [gross] payroll, except for the gross pay column.

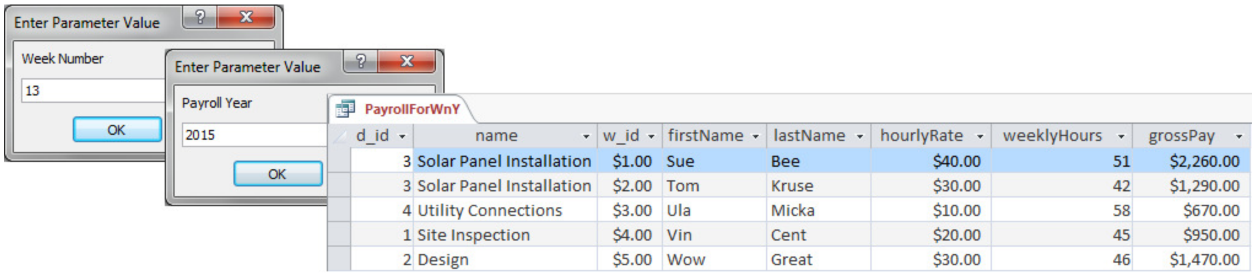


PayrollForWnY

PDWY: *, d_id, name, w_id, firstName, lastName, r, h

Field:	d_id	name	w_id	firstName	lastName	hourlyRate: r	weeklyHours: h	grossPay: Round([r]*([h]+(IIf([h]>40,[h],40)+IIf([h]>60,[h],60)-100)/2),2)
Table:	PDWY	PDWY	PDWY	PDWY	PDWY	PDWY	PDWY	
Sort:			Ascending					
Show:	✓	✓	✓	✓	✓	✓	✓	✓

Figure 15. The final query, `PayrollForWnY`, to compute the weekly gross payroll.



Enter Parameter Value — Week Number: 13 — OK

Enter Parameter Value — Payroll Year: 2015 — OK

PayrollForWnY

d_id	name	w_id	firstName	lastName	hourlyRate	weeklyHours	grossPay
3	Solar Panel Installation	$1.00	Sue	Bee	$40.00	51	$2,260.00
3	Solar Panel Installation	$2.00	Tom	Kruse	$30.00	42	$1,290.00
4	Utility Connections	$3.00	Ula	Micka	$10.00	58	$670.00
1	Site Inspection	$4.00	Vin	Cent	$20.00	45	$950.00
2	Design	$5.00	Wow	Great	$30.00	46	$1,470.00

Figure 16. The `PayrollForWnY` query's output (the final result).

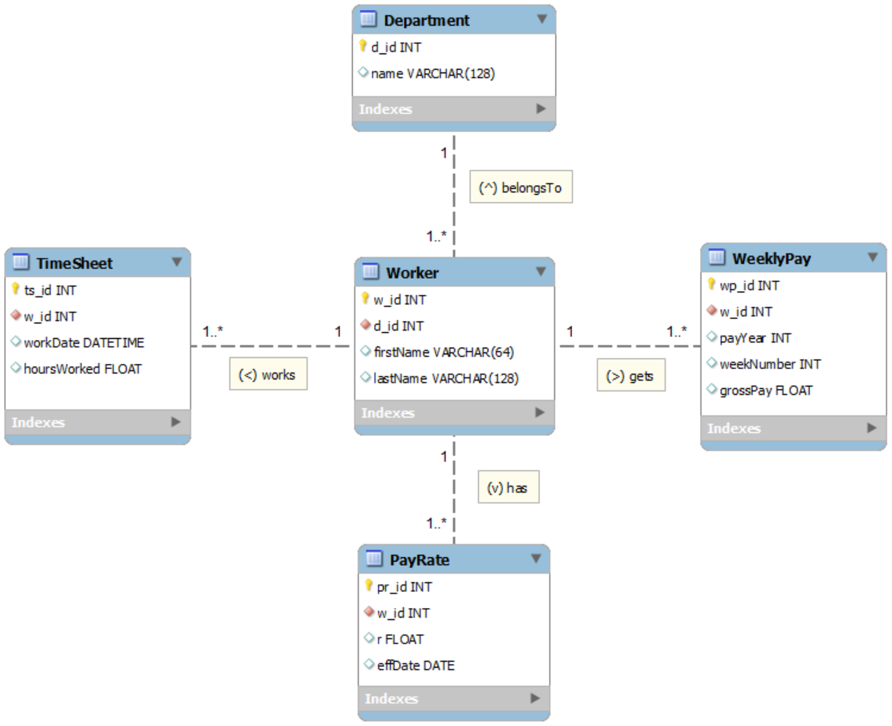
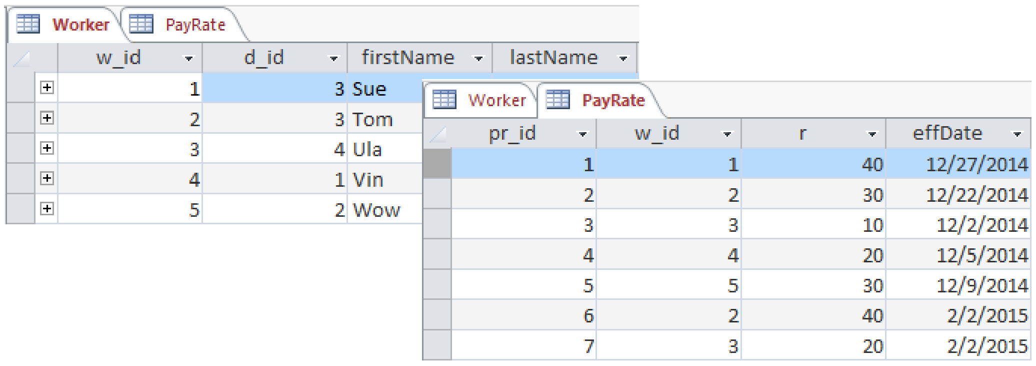


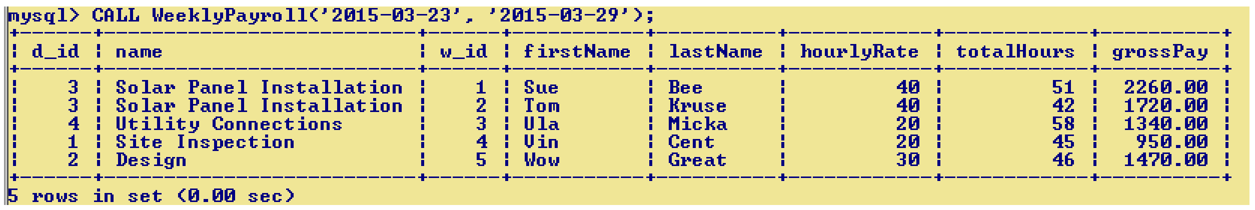
Figure 17. The updated ER diagram.



w_id	d_id	firstName	lastName
1	3	Sue	
2	3	Tom	
3	4	Ula	
4	1	Vin	
5	2	Wow	

pr_id	w_id	r	effDate
1	1	40	12/27/2014
2	2	30	12/22/2014
3	3	10	12/2/2014
4	4	20	12/5/2014
5	5	30	12/9/2014
6	2	40	2/2/2015
7	3	20	2/2/2015

Figure 18. A new table, `PayRate`, and modified table `Worker` (without column `hourlyPayRate`).



```
mysql> CALL WeeklyPayroll('2015-03-23', '2015-03-29');
+------+--------------------------+------+-----------+----------+------------+------------+-----------+
| d_id | name                     | w_id | firstName | lastName | hourlyRate | totalHours | grossPay  |
+------+--------------------------+------+-----------+----------+------------+------------+-----------+
|    3 | Solar Panel Installation |    1 | Sue       | Bee      |         40 |         51 |   2260.00 |
|    3 | Solar Panel Installation |    2 | Tom       | Kruse    |         40 |         42 |   1720.00 |
|    4 | Utility Connections      |    3 | Ula       | Micka    |         20 |         58 |   1340.00 |
|    1 | Site Inspection          |    4 | Vin       | Cent     |         20 |         45 |    950.00 |
|    2 | Design                   |    5 | Wow       | Great    |         30 |         46 |   1470.00 |
+------+--------------------------+------+-----------+----------+------------+------------+-----------+
5 rows in set (0.00 sec)
```

Figure 19. The result of calling the WeeklyPay stored procedure for week 13 of year 2015 (between dates '2015-03-23' and '2015-03-29').