Case Study of the Failures of Relational Database Management Systems

Team Number #1,
Kris Beckman, Bachelor of Science Information Systems (BSIS)
Jennifer Jamiel, Bachelor of Science Information Systems (BSIS)
Thaddeus Thomas, Bachelor of Science Information Systems (BSIS)
Deepti Venkatesh, Bachelor of Science Applied Computing (BSAPC)
Professor Donna Hoffman,
Information Systems 456 Database Systems Management,
School of Technology and Computing, City University of Seattle
kristina060585@cityuniversity.edu, jamieljennifer@cityuniversity.edu,
thomasthaddeus@cityuniversity.edu, d_974@cityuniversity.edu

Abstract

This paper will contain a case study of the failures of RDBMS (relational database management systems). Specifically, it will focus on what processes are restrictive to ensuring database integrity. Furthermore, it will show the methods that force atomicity, consistency, isolation, and durability (ACID) in operations that restrict the usability of RDBMS. Lastly, this case study will show how scaling vertically wastes enormous space and processing power when compounded across thousands of databases.

Keywords: RDBMS, ACID (atomicity, consistency, isolation, durability), CRUD (Create, Read, Update, Delete), MySQL, Data Management, SQLite

1. INTRODUCTION

The first thing constantly referenced when creating a database is reading past use cases. The reason is that database design is much more complicated than it appears. The process of setting up the database requires calculated steps. Add or remove entities, and the entire structure becomes unstable.

2. DATABASE DESIGN SUMMARY

In approaching the database design for our company employee database, we chose a relational data model due to the highly structured nature of employee data. In addition, high ACID guarantees are needed because of the sensitivity of data to avoid corruption and out-of-sync data. Finally, the relational model strikes the right balance because we will not need to scale thousands to millions of reads and writes, where NoSQL models shine. Specifically when the data is nonstructured (Coronel & Morris, 2018).

For this case study, the database is designed with three tables, initially: employees, jobs, and departments. The way they relate to each other can be seen in figure 1. We have also included stored procedures for manipulating data in a controlled, governed fashion. To work around the query performance bottlenecks of JOINS, we have created query-optimized VIEWS for many of our core reads. This simple model still gives elegant flexibility as employees can be managers with self-referential keys without introducing the fourth table. This model can be extended in the future if needed without a complete re-design of the database. For example, we could add an employee-type table with values such as individual contributor, manager, executive, board member, and foreign key into the employee table.

Related Work

Numerous studies have been written about the differences between relational databases and their NoSQL counterparts (Fowler, 2015). What this case study endeavors to support are where they both excel and where they fail. The database built for this project will be used for examples to show how the data can be oriented using multiple systems.

3. ENTITY RELATIONSHIP DIAGRAM

The following picture, figure 1, is the ERD (Entity Relationship Diagram). This diagram explains the data model & its built-in relations to form the core of a company employee database. Three tables populate the database.

Employee
The employee table of the database is where all the critical information is stored for employees. [EmployeeID] is the primary key and [JobID] is the foreign key.

Job
This table contains job details and salary compensation levels. [JobID] is the primary key in this table. [DeptID] is the foreign key. [CommPerc] is shorthand for a commission percentage.

Department
The Department table has the primary key [DeptID]. The foreign key on this table is the primary key on the job table [JobID].

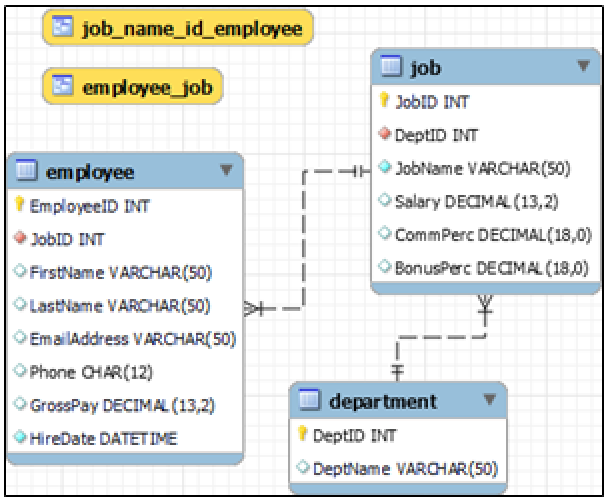

Figure 1-Entity Relationship Diagram

4. LITERATURE REVIEW

The reasons why a relational database might fail are numerous. Poorly written queries can corrupt a database quickly, especially if words like CASCADE are part of them (Petkovic, 2020, p.100-101). The literature review for this document is a collection of typical failures and the reasons why they occur. The intent is to be platform neutral to identify the root causes of why relational databases fail. Lastly, at the literature review's conclusion, a reading of best practices and insights will be presented to highlight where they still go right.

The largest corporations utilize RDBMS solutions in the world to power critical systems. SQL Databases have their pros and cons. One of the pros is the strong consistency models they implement. Strong consistency comes at the price of either Availability or Partition Tolerance. Care must be taken to ensure no data loss in disaster scenarios. There have been many examples of this through the years (Kapuya*,* 2013).

The first example is Magnolia*,* a once popular online bookmark-sharing website. On January 30, 2009, San Francisco suffered a catastrophic meltdown and a significant data loss. The site lost its primary store of user data and its backup. Due to hard drive corruption, the data was not successfully restored, and their CEO had to inform users. The issue took days and not hours to solve and had irrecoverable brand damage to the business. The main reason explained for the issue was the lack of redundancy for the data storage and backups, so when the primary drive was corrupted, both the backups and the data files were corrupt. The primary lesson learned was that when using RDBMS solutions such as MySQL, in a high transaction volume environment, you must ensure utilizing redundant storage solutions, consistent backup storage in a separate environment with redundancy there as well, and finally, a set of processes and tools to constantly test Disaster recovery scenarios, (Calore, 2009).

Large-scale web applications with simple schema such-as bookmarks need to favor Availability and Partition Tolerance and not consistency in the CAP Theorem model. NoSQL Databases are spread across multiple nodes with the ability to full multi-replica nodes so that even if many nodes fail, the likelihood of data loss is minimalized. A great showcase of this capability is Netflix, which uses Cassandra, one of the leading NoSQL Databases. They had to reboot multiple nodes in 2014; 22 did not come back, all were replaced, and there was 0 downtime (Avram, 2014).

The second database failure that caused an outage was GitHub. GitHub is the leading Code Storage and version control platform utilized by most major organizations worldwide. Microsoft currently owns GitHub (Kapuya, 2013).

From September 10 to 11th, 2012, in San Francisco, GitHub, which used MySQL Database Servers, had an outage that lasted 1 Hour and 46

Minutes. While it might not seem long, many organizations dependent on GitHub for day-to-day development were affected during a maintenance window to replace an old cluster of replicated MySQL Servers that were DRBD-Backed. They needed to do this because failover management in the old replication set required cold restarts and the new cluster would always have running nodes, so failovers become efficient (Newland, 2012).

The first outage for GitHub on the 10th occurred during the zero-downtime migration of MySQL Schema, where the live production load was so high that it caused the replication manager's health checks to fail to cause a chain reaction. On the 11th, the issue was caused by maintenance mode, which caused eventual inconsistencies in the data. Their team had to shut down the process, kill leaked events, and bring everything back up. The root cause of these outages was automated failovers in production while migrating servers. Automated failovers are great for availability but not for the primary database. GitHub put in processes to ensure that failover scenario only happens explicitly by a team member. In addition, GitHub added a database spare to improve availability models (Newland, 2012).

This is a good lesson in running RDBMS systems in large-scale scenarios that replication models can be complex due to the Consistency Constraints in RDBMS systems. GitHub requires ACID compliance due to its use-cases, so a NoSQL Database for core GitHub uses might not be possible.

While a non-relational model, such as document databases, could be utilized due to flexibility in Data modeling, we chose a relational model and an ACID compliant RDBMS due to strong transactional requirements due to data sensitivity. In addition, employee databases reside as part of a company's internal business systems, so usage levels will never be large enough to justify a NoSQL database. In addition, most core employee data can be stored in a single node with backup replicas, which our Database solution will support. In the future, we could supplement this database with a graph or document database for additional use-cases as needed, but this DB will still be treated as a Golden record/source of truth (Wayland, 2015). For these reasons, we need to favor consistency over multi-node scale. This need outweighs any of the shortfalls and reasons for failures of traditional RDBMS databases as specified earlier in this section, as all of those can be controlled through internal business processes within an organization. Large organizations such as Microsoft and Google still utilize RDBMS systems for core business & employee information internally.

5. METHODOLOGY

The definitive version of the database for this project was built in MySQL. It was initially built with SQLite, Postgres SQL, SQL Server, and CosmosDB. After customer requirements changed, the SQL queries were altered to conform to MySQL syntax. To build the employee database, we chose MySQL, a leading RDBMS utilized by numerous industry leaders. Next, the requirements were analyzed, and we built the data model based on the 2nd-Normal form to avoid data duplication (*Second Normal Form,* 2019).

The requirement was to have employees and their managers, their Job & Salary description, and the departments in which they exist. To do this, rather than duplicating data in the employee table, we separated Jobs & Salaries into their own table as multiple employees can have the same job title. We associated Jobs with Employees through a foreign key association and constraint.

An argument could be made that Jobs and Departments should be separate as two departments could theoretically have the same job but based on the requirements review, we consciously decided that Departments would act as a container of a family of related jobs. For example, the HR Department would not have an IT Related job, even though an IT Employee could help with work in the HR Department. We would have created department-specific schemas if we had known it was being constructed in a database other than SQLite. This quote from SQL Server 2019: A Beginners Guide describes precisely what a schema is and what purpose it serves.

> A *schema* is a database object that includes statements for creating tables, views, and user privileges. (You can think of a schema as a construct that collects several tables, corresponding views, and user privileges.) (Petkovic, 2020, p.102)

This will provide a good reporting and competence structure to the company utilizing this solution for their employee database. The model also gives sufficient flexibility if this needs to be changed in the future, as adding a link table or a foreign key to employees linking it to departments would not be hard to implement.

Once there was an agreed-upon model, we formalized it into the ERD diagram in figure 1. Utilizing the ERD diagram, we created scripts to build the database in MySQL and then the relevant Stored Procedures and Queries to validate that the database works as designed. The tables as shown in MySQL can be seen in Figure 2.

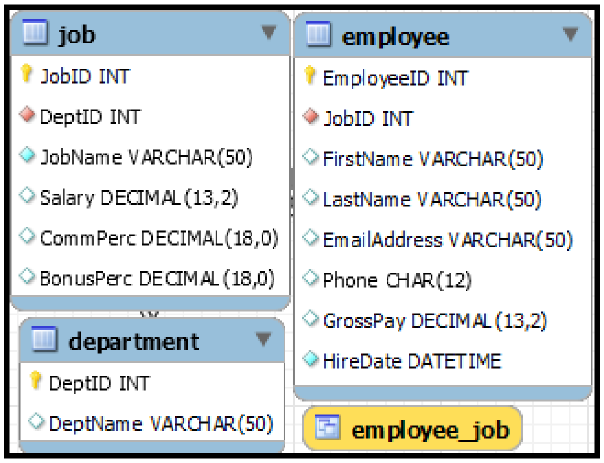

Figure 2

The following scripts were created in order of execution:

EmployeeDatabase.Sql
The main file contains the commands to create the database, the tables, the index for JobID for efficient querying, and required Views such as Job_Name_ID_Employee for common queries that need read efficiencies built into their design.

[UpdateEmployeeGrossPay.Sql]
Stored procedure to control the update of gross pay in the employee's table. A stored procedure allows for control and validation in the future of sensitive information updates. Additionally, teams can build additional stored procedures as necessary.

[PopulateData.sql]
Populates the employee database with test data to validate the correct setup. We built data themed around the sitcom office.

PopulateDataFailCheckConstraintSalary.sql
We want to validate that our check constraint on salary needing to be over 1000 is on and working. This file has an insert statement that SHOULD Fail if our database is set up correctly.

[Queries.sql]
All our required queries to highlight and validate our employee database as per requirements.

In summary, this has outlined our approach and methodologies in approaching the Employee Database need for the company.

6. SUMMARY

7. CONCLUSION

8. REFERENCES

Avram, A. (2014, October 28). *How Netflix Handled the Reboot of 218 Cassandra Nodes*. InfoQ. https://www.infoq.com/news/2014/10/netflix-cassandra/

Calore, M. (2009, January 30). *Magnolia Suffers Major Data Loss, Site Taken Offline*. WIRED. https://www.wired.com/2009/01/magnolia-suffer/

Coronel, C., & Morris, S. (2018). *Database Systems: Design, implementation, & management* (13th ed.). Cengage Learning.

Dušan Petković. (2020). *Microsoft SQL Server 2019 : a beginner's guide*. McGraw-Hill Education.

Fowler, A. (2015). *NoSQL For Dummies*. John Wiley and Sons.

Kapuya, A. (2013, March 21). *Infographic: MySQL Disasters in the Cloud*. Dzone.Com. https://dzone.com/articles/infographic-mysql-disasters

MySQL - Handling Duplicates. (n.d.). Www.tutorialspoint.com. Retrieved August 15, 2022

Newland, J. (2012, September 14). *GitHub availability this week*. The GitHub Blog. https://github.blog/2012-09-14-github-availability-this-week/

Petkovic, D. (2020). *Microsoft SQL Server 2019: A beginner's guide, seventh edition* (7th ed.). McGraw Hill.

Second Normal Form (2NF) - GeeksforGeeks. (2019, July 31). GeeksforGeeks.

Silberschatz, A., Korth, H. F., & S Sudarshan. (2006). *Database system concepts*. McGraw-Hill.

Wayland, J. (2015, May 7). *What is a Golden Record in MDM*. Informatica. https://www.informatica.com/blogs/golden-record.html

Zhang, W. (2012). *A Suite of Case Studies in Relational Database Design* (Doctoral dissertation).

Appendices

Tasks

#1. Write a query to display the names (first_name, last_name) using alias name "First Name", "Last Name"

```
SELECT
    FirstName AS 'First Name',
    LastName AS 'Last Name'
FROM employee;
```

#2. Write a query to get unique department ID from employee table.

```
SELECT DISTINCT DeptID
FROM Job;
```

#3. Write a query to get all employee details from the employee table order by first name, descending.

```
SELECT *
FROM employee
ORDER BY FirstName DESC;
```

#4. Write a query to get the names (first_name, last_name), salary, PF of all the employees (PF is calculated as 12% of salary).

```
SELECT (e.FirstName + e.LastName) AS Names, j.Salary, j.CommPerc,
j.BonusPerc, e.GrossPay, j.Salary*.15 as PF
FROM Employee e
INNER JOIN Job j
ON j.JobID = e.JobID
WHERE j.Salary > 10000
ORDER BY j.salary ASC;
```

#5. Write a query to get the employee ID, names (first_name, last_name), salary in ascending order of salary.

```
SELECT e.EmployeeID, (e.FirstName || ' ' || e.LastName) AS Names,
j.Salary
FROM Employee e
INNER JOIN Job j
ON j.JobID = e.JobID
ORDER BY j.Salary ASC;
```

#6. Write a query to get the total salaries payable to employees.

```
SELECT SUM(j.Salary)
FROM Employee e
INNER JOIN Job j
ON j.JobID = e.JobID;
```

#7. Write a query to get the maximum and minimum salary from the employees' table.

```
SELECT
    MAX(Salary) AS 'Maximum',
    MIN(Salary) AS 'Minimum'
FROM Job;

SELECT
    MAX(GrossPay) AS 'Maximum',
    MIN(GrossPay) AS 'Minimum'
```

```
        FROM Employee;
```

#8 Write a query to get the average salary and number of employees in the employees' table

```
        SELECT
            ROUND(AVG(j.Salary)) AS avg_salary,
            COUNT(DISTINCT e.EmployeeID) AS 'Total Employees'
        FROM Employee e
        INNER JOIN Job j
        ON j.JobID = e.JobID;

-- Database for IS456 Team Project
CREATE DATABASE IF NOT EXISTS EmployeeDB;

USE EmployeeDB;


/* Drop tables in reverse order for Foreign Key Constraints*/
DROP TABLE IF EXISTS Employee;
DROP TABLE IF EXISTS Job;
DROP TABLE IF EXISTS Department;

-- Table: Department
CREATE TABLE IF NOT EXISTS Department
(   DeptID INT NOT NULL,
    DeptName NVARCHAR(50) NULL,
    CONSTRAINT PK_Department PRIMARY KEY (DeptID ASC)
);

-- Table: Job
CREATE TABLE IF NOT EXISTS Job
(
    JobID INT NOT NULL,
    DeptID INT NOT NULL,
    JobName NVARCHAR(50) NOT NULL,
    Salary DECIMAL(13,2) NULL,
    CommPerc DECIMAL(18, 0) NULL,
    BonusPerc DECIMAL(18, 0) NULL,
    CONSTRAINT PK_Job PRIMARY KEY(JobID),
    CONSTRAINT  FK_Job_Department  FOREIGN  KEY  (DeptID)  REFERENCES
Department(DeptID)
);

-- Table: Employee
CREATE TABLE IF NOT EXISTS Employee
(
    EmployeeID INT NOT NULL,
    JobID INT NOT NULL,
    FirstName NVARCHAR(50) NULL,
    LastName NVARCHAR(50) NULL,
    EmailAddress NVARCHAR(50) NULL,
    Phone CHAR(12) NULL,
    GrossPay DECIMAL(13,2) NULL,
    HireDate DATETIME DEFAULT CURRENT_TIMESTAMP NOT NULL,
    CONSTRAINT PK_Employee PRIMARY KEY(EmployeeID),
    CONSTRAINT   FK_Employee_Job   FOREIGN   KEY   (JobID)   REFERENCES
Job(JobID),
    CONSTRAINT `GrossPay_chk` CHECK ((`GrossPay` > 1000))
);

-- Index: JobID
ALTER TABLE `Job` ADD INDEX `JobID` (`JobID`);

-- Views for IS456 Team Project
DROP VIEW IF EXISTS Employee_Job;

CREATE VIEW Employee_Job AS
SELECT
    CONCAT(e.FirstName,' ', e.LastName) AS full_name,
    j.JobName AS job_title
FROM Employee e
JOIN
Job j ON j.JobID = e.JobID
GROUP BY full_name, job_title;

-- View: Job_Name_ID_Employee
DROP VIEW IF EXISTS Job_Name_ID_Employee;

CREATE VIEW Job_Name_ID_Employee  AS
SELECT
    j.JobID AS job_code,
    j.JobName AS Job,
    CONCAT(e.FirstName,' ', e.LastName) AS employee_name
FROM Job j
JOIN Employee e
  ON j.JobID = e.JobID;

-- DATA To populate db
USE EmployeeDB;
```

```
/* Clear data in reverse order due to foriegn key constraints */
DELETE FROM Employee WHERE EmployeeID > 0;
DELETE FROM Job WHERE JobID > 0;
DELETE FROM Department WHERE DeptID > 0;

/* Populate Departments first */

INSERT INTO Department (DeptID, DeptName)
VALUES (1, 'Corporate');
INSERT INTO Department (DeptID, DeptName)
VALUES (2, 'Regional Management');
INSERT INTO Department (DeptID, DeptName)
VALUES (3, 'HR');
INSERT INTO Department (DeptID, DeptName)
VALUES (4, 'Accounting');
INSERT INTO Department (DeptID, DeptName)
VALUES (5, 'Qwality Assurance');
INSERT INTO Department (DeptID, DeptName)
VALUES (6, 'Customer Service');
INSERT INTO Department (DeptID, DeptName)
VALUES (7, 'Sales');
INSERT INTO Department (DeptID, DeptName)
VALUES (8, 'IT');
INSERT INTO Department (DeptID, DeptName)
VALUES (9, 'Reception');
INSERT INTO Department (DeptID, DeptName)
VALUES (10, 'Office Management');

/* Populate Jobs Next */

INSERT INTO Job (JobID, DeptID, JobName, Salary, CommPerc, BonusPerc)
VALUES (1, 1, 'CEO', 300000.00, NULL, 40.0);
INSERT INTO Job (JobID, DeptID, JobName, Salary, CommPerc, BonusPerc)
VALUES (2, 1, 'CFO', 280000.00, NULL, 40.0);
INSERT INTO Job (JobID, DeptID, JobName, Salary, CommPerc, BonusPerc)
VALUES (3, 2, 'General Manager', 180000.00, NULL, 30.0);
INSERT INTO Job (JobID, DeptID, JobName, Salary, CommPerc, BonusPerc)
VALUES (4, 2, 'Assistant to the General Manager', 130000.00, NULL, 15.0);
INSERT INTO Job (JobID, DeptID, JobName, Salary, CommPerc, BonusPerc)
VALUES (5, 3, 'HR Rep', 80000.00, NULL, 10.0);
INSERT INTO Job (JobID, DeptID, JobName, Salary, CommPerc, BonusPerc)
VALUES (6, 4, 'Accountant', 90000.00, NULL, 10.0);
INSERT INTO Job (JobID, DeptID, JobName, Salary, CommPerc, BonusPerc)
VALUES (7, 5, 'QA Engineer', 80000.00, NULL, 10.0);
```

```
INSERT INTO Job (JobID, DeptID, JobName, Salary, CommPerc, BonusPerc)
VALUES (8, 6, 'Customer Success Manager', 70000.00, NULL, 10.0);
INSERT INTO Job (JobID, DeptID, JobName, Salary, CommPerc, BonusPerc)
VALUES (9, 7, 'Sales Manager', 70000.00, 30.0, NULL);
INSERT INTO Job (JobID, DeptID, JobName, Salary, CommPerc, BonusPerc)
VALUES (10, 8, 'IT Specialist', 95000.00, NULL, 10.0);
INSERT INTO Job (JobID, DeptID, JobName, Salary, CommPerc, BonusPerc)
VALUES (11, 9, 'Receptionist', 55000.00, NULL, 10.0);
INSERT INTO Job (JobID, DeptID, JobName, Salary, CommPerc, BonusPerc)
VALUES (12, 10, 'Office Manager', 60000.00, NULL, 10.0);

/* Employee Table Last */

INSERT INTO Employee (EmployeeID, JobID, FirstName, LastName,
EmailAddress, Phone, GrossPay, HireDate)
VALUES (1, 2, 'David', 'Wallace', 'david.wallace@office.com', '5555555000',
380000.00, '2006-03-03 01:15:00');
INSERT INTO Employee (EmployeeID, JobID, FirstName, LastName,
EmailAddress, Phone, GrossPay, HireDate)
VALUES (2, 3, 'Michael', 'Scott', 'michael.scott@office.com', '5555555000',
220000.00, '2006-03-03 01:15:00');
INSERT INTO Employee (EmployeeID, JobID, FirstName, LastName,
EmailAddress, Phone, GrossPay, HireDate)
VALUES (3, 4, 'Dwight', 'Schrute', 'dwight.schrute@office.com', '5555555000',
145000.00, '2006-03-03 01:15:00');
INSERT INTO Employee (EmployeeID, JobID, FirstName, LastName,
EmailAddress, Phone, GrossPay, HireDate)
VALUES (4, 5, 'Toby', 'Flenderson', 'toby.flenderson@office.com',
'5555555000', 85000.00, '2006-03-03 01:15:00');
INSERT INTO Employee (EmployeeID, JobID, FirstName, LastName,
EmailAddress, Phone, GrossPay, HireDate)
VALUES (5, 6, 'Angela', 'Martin', 'angela.martin@office.com', '5555555000',
94000.00, '2006-03-03 01:15:00');
INSERT INTO Employee (EmployeeID, JobID, FirstName, LastName,
EmailAddress, Phone, GrossPay, HireDate)
VALUES (6, 7, 'Creed', 'Bratton', 'creed.bratton@office.com', '5555555000',
85000.00, '2006-03-03 01:15:00');
INSERT INTO Employee (EmployeeID, JobID, FirstName, LastName,
EmailAddress, Phone, GrossPay, HireDate)
VALUES (7, 8, 'Kelly', 'Kapoor', 'kelly.kapoor@office.com', '5555555000',
73000.00, '2006-03-03 01:15:00');
INSERT INTO Employee (EmployeeID, JobID, FirstName, LastName,
EmailAddress, Phone, GrossPay, HireDate)
```

```
VALUES (8, 9, 'Jim', 'Halpert', 'jim.halpert@office.com', '5555555000',
97000.00, '2006-03-03 01:15:00');
INSERT INTO Employee (EmployeeID, JobID, FirstName, LastName,
EmailAddress, Phone, GrossPay, HireDate)
VALUES (9, 11, 'Erin', 'Hannon', 'erin.hannon@office.com', '5555555000',
55000.00, '2006-03-03 01:15:00');
INSERT INTO Employee (EmployeeID, JobID, FirstName, LastName,
EmailAddress, Phone, GrossPay, HireDate)
VALUES (10, 12, 'Pam', 'Halpert', 'pam.halper@office.com', '5555555000',
70000.00, '2006-03-03 01:15:00');



-- Stored procedure for modifying data
USE EmployeeDB;

DROP PROCEDURE IF EXISTS UpdateEmployeeGrossPay;

DELIMITER //

CREATE PROCEDURE UpdateEmployeeGrossPay (IN ID INT, IN NewGrossPay
DECIMAL(13,2))
BEGIN

UPDATE Employee
SET GrossPay = NewGrossPay
WHERE EmployeeID = ID;

END//

DELIMITER ;

CALL UpdateEmployeeGrossPay(3, 99383.22);

SELECT * FROM Employee WHERE EmployeeID = 3;

-- Validate check constraint as required
USE EmployeeDB;

# Gross Pay must be over 1000
INSERT INTO Employee (EmployeeID, JobID, FirstName, LastName,
EmailAddress, Phone, GrossPay, HireDate)
VALUES (11, 2, 'Jan', 'Levinson', 'jan.levinson@office.com', '5555555000',
500.00, '2006-03-03 01:15:00');
```