CENG352 Written Assignment 1

Mustafa Ozan Alpay

28/04/2021

1 XML and JSON

1.1 XML

```xml
<X>
    <A>
        <A>one</A>
        <B>
            <B>two</B>
            <B>three</B>
        </B>
        <C>four</C>
    </A>
    <A>
        <B>
            <A>five</A>
            <A>six</A>
        </B>
        <C>seven</C>
    </A>
</X>
```

1. /X/A/C/text() ⇒ four, seven
2. /X/A/*/text() ⇒ one, four, seven
3. //A[B/A]/C/text() ⇒ seven
4. /X/A[A]/B/*/text() ⇒ two, three
5. //A/B/*/text() ⇒ two, three, five, six
6. /X/A[A][C]/B/*/text() ⇒ two, three

1.2 JSON

```json
{
  "suppliers": [
    {
      "sid": 101,
      "sname": "Acme",
      "address": "123 Main",
      "catalog": [
        {
          "pid": 92,
          "pname": "handle",
          "color": "Green",
          "cost": 5.21
        }
      ]
    },
    {
      "sid": 102,
      "sname": "Ace",
      "address": "456 Lake",
      "catalog": [
        {
          "pid": 92,
          "pname": "handle",
          "color": "Green",
          "cost": 6.5
        },
        {
          "pid": 93,
          "pname": "gasket",
          "color": "Red",
          "cost": 65.99
        }
      ]
    },
    {
      "sid": 103,
      "sname": "Figaro",
      "address": "678 First",
      "catalog": []
    }
  ],
  "parts": [
    {
      "pid": 90,
      "pname": "bumper",
      "color": "Red"
    },
    {
      "pid": 91,
      "pname": "caliper",
      "color": "Blue"
    },
    {
      "pid": 92,
      "pname": "handle",
      "color": "Green"
    },
    {
      "pid": 93,
      "pname": "gasket",
      "color": "Red"
    }
  ]
}
```

The model above does not avoid redundancy since the details of parts are contained within the Suppliers object as well. In order to avoid redundancy, removing part details from suppliers and storing only the `pid` and `cost` might be considered. The version with the mentioned changes can be found below.

```json
{
    "suppliers": [
        {
            "sid": 101,
            "sname": "Acme",
            "address": "123 Main",
            "catalog": [
                {
                    "pid": 92,
                    "cost": 5.21
                }
            ]
        },
        {
            "sid": 102,
            "sname": "Ace",
            "address": "456 Lake",
            "catalog": [
                {
                    "pid": 92,
                    "cost": 6.5
                },
                {
                    "pid": 93,
                    "cost": 65.99
                }
            ]
        },
        {
            "sid": 103,
            "sname": "Figaro",
            "address": "678 First",
            "catalog": []
        }
    ],
    "parts": [
        {
            "pid": 90,
            "pname": "bumper",
            "color": "Red"
        },
        {
            "pid": 91,
            "pname": "caliper",
            "color": "Blue"
        },
        {
            "pid": 92,
            "pname": "handle",
            "color": "Green"
        },
        {
            "pid": 93,
            "pname": "gasket",
            "color": "Red"
        }
    ]
}
```

2 Database Design

2.1 BCNF Decomposition

1. Conference (PaperNo, FirstAuthorNo, AuthorNo, AuthorName, AuthorEmail, AuthorAddress, PaperTitle, PaperAbstract, PaperStatus, ReviewerNo, ReviewerName, ReviewerEmail, Commments, ProgramComm, ReviewDate, Rating, ReviewerAddress) violates BCNF.

2. Split Conference into Author (AuthorNo, AuthorEmail, AuthorAddress, AuthorName) and Conference (PaperNo, FirstAuthorNo, AuthorNo, PaperTitle, PaperAbstract, PaperStatus, ReviewerNo, ReviewerName, ReviewerEmail, Commments, ProgramComm, ReviewDate, Rating, ReviewerAddress). Conference violates BCNF.

3. Split Conference into Paper (PaperNo, PaperTitle, PaperAbstract, PaperStatus, FirstAuthorNo) and Conference (PaperNo, AuthorNo, ReviewerNo, ReviewerName, ReviewerEmail, Commments, ProgramComm, ReviewDate, Rating, ReviewerAddress). Conference violates BCNF.

4. Split Conference into Reviewer (ReviewerNo, ReviewerEmail, ReviewerName, ReviewerAddress) and Conference (PaperNo, AuthorNo, ReviewerNo, Commments, ProgramComm, ReviewDate, Rating). Conference does not violate BCNF.

5. Since we reached a state where no more violations, we can say that the decomposition is complete and the schema is in Boyce-Codd Normal Form.

- Author (AuthorNo *UNIQUE*, AuthorEmail *UNIQUE*, AuthorAddress, AuthorName)
- Paper (PaperNo *UNIQUE*, PaperTitle, PaperAbstract, PaperStatus, FirstAuthorNo)
- Reviewer (ReviewerNo *UNIQUE*, ReviewerEmail *UNIQUE*, ReviewerName, ReviewerAddress)
- Conference (PaperNo *FK*, ReviewerNo *FK*, AuthorNo *FK*, ReviewDate, Rating, Comments, ProgramComm)

Since we preserved unique fields and the decompositions are made according to the keys, the BCNF Decomposition above is lossless. In order to see that, we can simply join the relations and see that due to the foreign keys, primary keys and unique fields, we can get the tuples that has the same entries as the original version without any new or missing ones.

However, it is not dependency preserving. For example, let's consider the functional dependency "PaperNo → FirstAuthorNo". If we check the final Conference relation, we can see that this dependency is not met. Therefore it is not dependency preserving.

2.2 3NF Decomposition

1. Minimal Cover

   (a) Rewrite functional dependencies as one attribute per right hand side.

   - AC → B
   - AC → G
   - AC → H
   - D → E
   - G → B
   - E → F
   - E → K
   - FD → K
   - ADF → C
   - H → B

- H $\to$ G
- H $\to$ H

(b) Eliminate redundant attributes.

- AC $\to$ B
- AC $\to$ G
- AC $\to$ H
- D $\to$ E
- G $\to$ B
- E $\to$ F
- E $\to$ K
- D $\to$ K
- AD $\to$ C
- H $\to$ B
- H $\to$ G

(c) Delete redundant functional dependencies.

- AC $\to$ H
- D $\to$ E
- G $\to$ B
- E $\to$ F
- E $\to$ K
- AD $\to$ C
- H $\to$ G

2. Decomposition into 3NF

(a) Compute a minimal cover U.

(b) Partition U into sets such that left hand side of all elements of U_i are the same.

- $U_1 = \{\text{AC} \to \text{H}\}$
- $U_2 = \{\text{D} \to \text{E}\}$
- $U_3 = \{\text{G} \to \text{B}\}$
- $U_4 = \{\text{E} \to \text{F}, \text{E} \to \text{K}\}$
- $U_5 = \{\text{AD} \to \text{C}\}$
- $U_6 = \{\text{H} \to \text{G}\}$

(c) For each U_i form schema $R_i = (R_i, U_i)$.

- $R_1 = (\text{ACH}; \text{AC} \to \text{H})$
- $R_2 = (\text{DE}; \text{D} \to \text{E})$
- $R_3 = (\text{GB}; \text{G} \to \text{B})$
- $R_4 = (\text{EFK}; \text{E} \to \text{F}, \text{E} \to \text{K})$
- $R_5 = (\text{ADC}; \text{AD} \to \text{C})$
- $R_6 = (\text{HG}; \text{H} \to \text{G})$

(d) We need to check if any R_i is a superkey of R, in that case we can conclude the decomposition. If that fails, we would need $R_0 = (R_0, \{\})$ as another schema. Since AD is a candidate key, and since R_5 includes AD, we can conclude the decomposition.

(e) $R_1 = (\text{ACH}), R_2 = (\text{DE}), R_3 = (\text{GB}), R_4 = (\text{EFK}), R_5 = (\text{ADC}), R_6 = (\text{HG})$

2.3 Finding Dependencies

1. Functional Dependencies

   - A → B
   - B → A
   - C → D
   - D → C
   - F → G
   - G → F

   Successful Queries

```sql
SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.A = d2.A
WHERE d1.B != d2.B;
-- Returns 0, therefore A -> B.

SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.B = d2.B
WHERE d1.A != d2.A;
-- Returns 0, therefore B -> A.

SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.C = d2.C
WHERE d1.D != d2.D;
-- Returns 0, therefore C -> D.

SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.D = d2.D
WHERE d1.C != d2.C;
-- Returns 0, therefore D -> C.

SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.F = d2.F
WHERE d1.G != d2.G;
-- Returns 0, therefore F -> G.

SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.G = d2.G
WHERE d1.F != d2.F;
-- Returns 0, therefore G -> F.
```

   Unsuccessful Queries

```sql
SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.A = d2.A
WHERE d1.C != d2.C;
-- Returns 20047500, A -> C does not hold.
-- Since C -> D, there is no need to check A -> D. It will not hold either.
-- Since B -> A, there is no need to check B -> D or B -> C. They will not hold either.

SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.A = d2.A
WHERE d1.E != d2.E;
-- Returns 20231402, A -> E does not hold.
-- Since B -> A, there is no need to check B -> E. It will not hold either.

SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.A = d2.A
WHERE d1.F != d2.F;
-- Returns 18900000, A -> F does not hold.
-- Since F -> G, there is no need to check A -> G. It will not hold either.
-- Since B -> A, there is no need to check B -> F or B -> G. They will not hold either.

SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.C = d2.C
WHERE d1.A != d2.A;
-- Returns 1620000, C -> A does not hold.
-- Since A -> B, there is no need to check C -> B. It will not hold either.
-- Since D -> C, there is no need to check D -> B or D -> A. They will not hold either.

SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.C = d2.C
WHERE d1.E != d2.E;
-- Returns 1808524, C -> E does not hold.
-- Since D -> C, there is no need to check D -> E. It will not hold either.

SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.C = d2.C
WHERE d1.F != d2.F;
-- Returns 1701000, C -> F does not hold.
-- Since F -> G, there is no need to check C -> G. It will not hold either.
-- Since D -> C, there is no need to check D -> F or D -> G. They will not hold either.

SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.E = d2.E
WHERE d1.A != d2.A;
-- Returns 40692, E -> A does not hold.
-- Since A -> B, there is no need to check E -> B. It will not hold either.

SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.E = d2.E
WHERE d1.C != d2.C;
-- Returns 45314, E -> C does not hold.
-- Since C -> D, there is no need to check E -> D. It will not hold either.

SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.E = d2.E
```

```sql
WHERE d1.F != d2.F;
-- Returns 42666, E -> F does not hold.
-- Since F -> G, there is no need to check E -> G. It will not hold either.

SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.F = d2.F
WHERE d1.A != d2.A;
-- Returns 10800000, F -> A does not hold.
-- Since A -> B, there is no need to check F -> B. It will not hold either.
-- Since F -> G, there is no need to check G -> A or G -> B. They will not hold either.

SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.F = d2.F
WHERE d1.C != d2.C;
-- Returns 12028500, F -> C does not hold.
-- Since C -> D, there is no need to check F -> D. It will not hold either.
-- Since F -> G, there is no need to check G -> C or G -> D. They will not hold either.

SELECT COUNT(*)
FROM data d1
JOIN data d2
ON d1.F = d2.F
WHERE d1.E != d2.E;
-- Returns 12133376, F -> E does not hold.
-- Since F -> G, there is no need to check G -> E. It will not hold either.
```

2. SQL Statements for creating normalized tables

```sql
CREATE TABLE r1 (
    A varchar(1),
    B varchar(7),
    PRIMARY KEY (A)
);

CREATE TABLE r2 (
    C int,
    D varchar(16),
    PRIMARY KEY (C)
);

CREATE TABLE r3 (
    F int,
    G varchar(16),
    PRIMARY KEY (F)
);

CREATE TABLE r4 (
    A varchar(1),
    C int,
    E int,
    F int,
    PRIMARY KEY (A, C, E, F),
    FOREIGN KEY (A) REFERENCES r1(A),
    FOREIGN KEY (C) REFERENCES r2(C),
    FOREIGN KEY (F) REFERENCES r3(F)
);
```

3. SQL Statements to load contents of the tables

```sql
-- Create temporary table
CREATE TABLE data (
    A varchar(1),
    B varchar(7),
    C int,
    D varchar(16),
    E int,
    F int,
    G varchar(16)
);

-- Import data to the temporary table
\copy data(A, B, C, D, E, F, G) FROM '//home/ozan/github/ceng352-wa1/sample.csv' with (format csv,
↪  header true, delimiter ',');

-- Insert data into normalized tables

INSERT INTO r1 (A, B)
SELECT DISTINCT A, B
FROM data;

INSERT INTO r2 (C, D)
SELECT DISTINCT C, D
FROM data;

INSERT INTO r3 (F, G)
SELECT DISTINCT F, G
FROM data;

INSERT INTO r4 (A, C, E, F)
SELECT DISTINCT A, C, E, F
FROM data;

-- Drop temporary table

DROP TABLE data;
```

3 SQL DDL

```sql
CREATE TABLE Customer (
    CustNo varchar(32) NOT NULL,
    CustFirstName varchar(32),
    CustLastName varchar(32),
    CustCity varchar(32),
    CustState varchar(4),
    CustZip varchar(16),
    CustBal float,
    PRIMARY KEY (CustNo)
);

CREATE TABLE Employee (
    EmpNo varchar(32) NOT NULL,
    EmpFirstName varchar(32),
    EmpLastName varchar(32),
    EmpPhone varchar(32),
    EmpEmail varchar(64),
    EmpDeptName varchar(32),
    EmpStatus varchar(32),
    EmpSalary float,
    supervisor varchar(32) NOT NULL DEFAULT '007',
    PRIMARY KEY (EmpNo),
    FOREIGN KEY (supervisor) REFERENCES Employee(EmpNo) ON DELETE SET DEFAULT,
    CHECK (EmpEmail NOT LIKE CONCAT('%', EmpFirstName, '%')),
    CHECK (EmpEmail NOT LIKE CONCAT('%', EmpLastName, '%'))
);

CREATE TABLE Product (
    ProdNo varchar(32) NOT NULL,
    ProdName varchar(32),
    ProdPrice float,
    ProdShipDate timestamp,
    PRIMARY KEY (ProdNo)
);

CREATE TABLE "Order" (
    OrdNo varchar(32) NOT NULL,
    CustNo varchar(32) NOT NULL,
    EmpNo varchar(32),
    OrdDate timestamp,
    OrdName varchar(32),
    OrdCity varchar(32),
    OrdZip varchar(16),
    PRIMARY KEY (OrdNo),
    FOREIGN KEY (CustNo) REFERENCES Customer(CustNo) ON DELETE CASCADE,
    FOREIGN KEY (EmpNo) REFERENCES Employee(EmpNo) ON DELETE SET NULL,
    CHECK (OrdName LIKE CONCAT('%', OrdCity, '%'))
);

CREATE TABLE Contains (
    OrdNo varchar(32) NOT NULL,
    ProdNo varchar(32) NOT NULL,
    Qty float,
    PRIMARY KEY (OrdNo, ProdNo),
    FOREIGN KEY (OrdNo) REFERENCES "Order"(OrdNo) ON DELETE CASCADE,
    FOREIGN KEY (ProdNo) REFERENCES Product(ProdNo) ON DELETE CASCADE,
    CHECK (Qty >= 3)
);

CREATE ASSERTION CheckQuantity CHECK (
    NOT EXISTS (SELECT COUNT(c.Qty)
    FROM Contains c
    JOIN "Order" o
    ON c.OrdNo = o.OrdNo
    GROUP BY o.OrdNo
    HAVING COUNT(c.Qty) < 30)
);

CREATE TRIGGER SalaryChange
AFTER UPDATE OF EmpSalary ON Employee
REFERENCING
OLD ROW AS OldTuple
NEW ROW AS NewTuple
FOR EACH ROW
WHEN (NewTuple.EmpSalary - OldTuple.EmpSalary >  0.15 * OldTuple.EmpSalary)
UPDATE Employee
SET EmpStatus = 'Successful'
WHERE EmpNo = OldTuple.EmpNo;

-- To allow insertion to the Employee table, I created the following insertion statement as well.
INSERT INTO Employee
VALUES ('007', 'Default', 'Supervisor', '', '', '', '', 0, '007');
```