CSE532 Project Report

Credit Card Database
via the Object-Oriented Extension of SQL

Author Name, #SBU ID

October 30, 2015

1 Database Design

Something of design

1.1 Entity-Relationship Design

After a series of consideration, we decided the final Entity-Relationship Model as Figure 1.

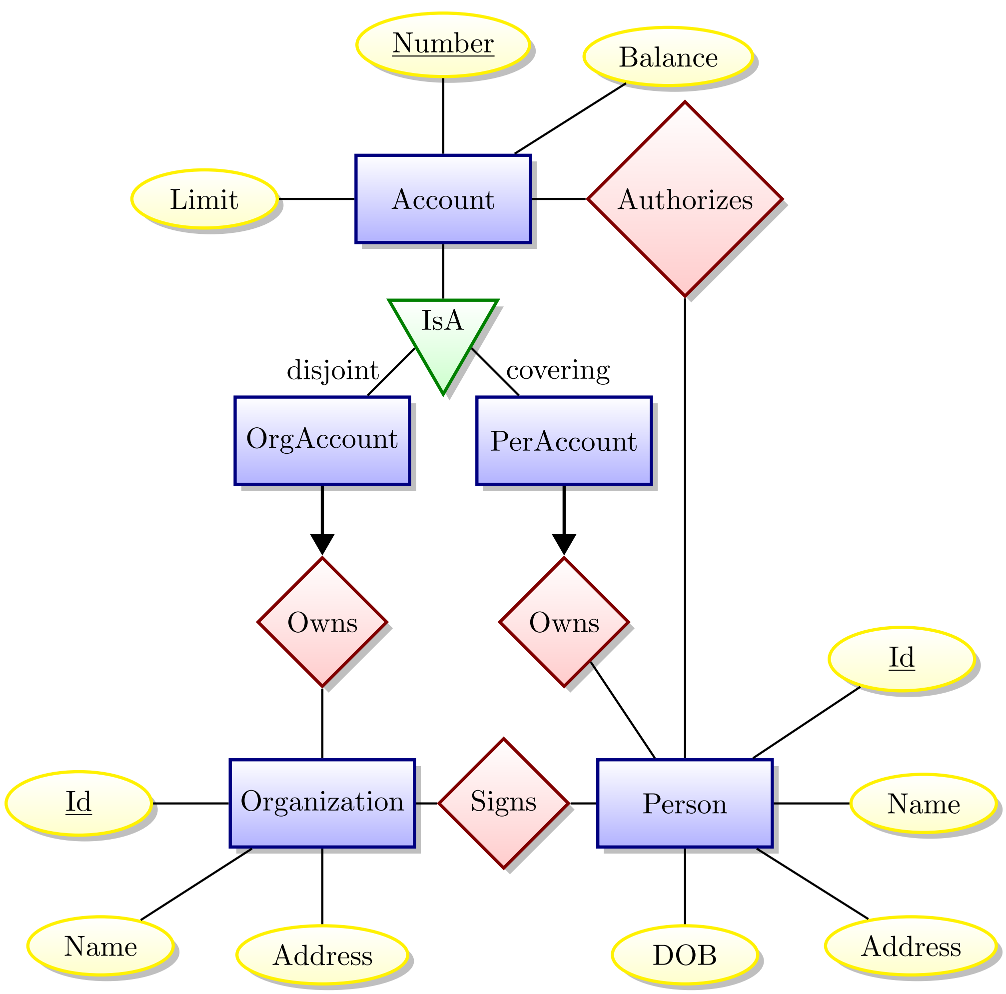


Figure 1: E-R diagram for CCDB

1.1.1 Brief explanation of E-R design

Since the owner of a credit card account can be either a person or an organization, we split the accounts into two types: Personal Account and Organizational Account. This partition on Account satifies the disjointness constraint and covering constraint.

After splitting the Account entity type into PerAccount and OrgAccount, we also split the relationship "Owns". But we don't need extra table to store these relationships. In fact, every account can have just one owner, no more and no less. We draw an arrow pointing in the direction of the relationship's diamond in very thick lines. We know that account Number is a key of PerAccount and also of PersonOwns. Thus, we can merge the attributes of PersonOwns into PerAccount. It is guaranteed that each PerAccount tuple has exactly one corresponding PersonOwns tuple, so no redundancy is created. Then we can concatenate PerAccount and PersonOwns tuples which have same account number, and the same goes to OrgAccount and OrganizationOwns.

The relationship Signs and Authorizes are many-to-many relationships. We have to create tables to store the relation, and define proper foreign keys.

1.1.2 Relational Database Schemas

Now we can convert the E-R diagram to relational database schemas. Noting that here we assume there is no inheritance and other object-oriented features. We keep Person and Organization entities. For account type hierarchy, we represent the IsA relationship in general representation. That means we create Account table stroing all information except owner. And table PerAccount and OrgAccount both have a foreign key pointing to Account and another foreign key pointing to the corresponding owner. The relationship Owns has been merged into PerAccount and OrgAccount. The SQL CREATE statements are as follows.

```
CREATE TABLE Person (
    Id      CHAR(20),
    Name    CHAR(20),
    Address CHAR(50),
    Dob     DATE,
    PRIMARY KEY (Id) )

CREATE TABLE Organization (
    Id      CHAR(20),
    Name    CHAR(20),
    Address CHAR(50)
    PRIMARY KEY (Id) )

CREATE TABLE Account (
    Number  CHAR(20),
    Balance DECIMAL,
    Limit   DECIMAL,
    PRIMARY KEY (Number) )

CREATE TABLE PerAccount (
    Number  CHAR(20),
    Owner   CHAR(20) NOT NULL,
    PRIMARY KEY (Number),
    FOREIGN KEY (Number) REFERENCES Account(Number),
    FOREIGN KEY (Owner) REFERENCES Person(Id) )

CREATE TABLE OrgAccount (
```

```
    Number  CHAR(20),
    Owner   CHAR(20) NOT NULL,
    PRIMARY KEY (Number),
    FOREIGN KEY (Number) REFERENCES Account(Number),
    FOREIGN KEY (Owner) REFERENCES Organization(Id) )

CREATE TABLE Signs (
    Pid CHAR(20),
    Oid CHAR(20),
    PRIMARY KEY (Pid, Oid),
    FOREIGN KEY (Pid) REFERENCES Person(Id),
    FOREIGN KEY (Oid) REFERENCES Organization(Id) )

CREATE TABLE Authorizes (
    Number CHAR(20),
    Pid    CHAR(20),
    PRIMARY KEY (Number, Pid),
    FOREIGN KEY (Number) REFERENCES Account(Number),
    FOREIGN KEY (Pid) REFERENCES Person(Id) )
```

1.2 Object-relational Design

The object-relational database concept provides us some very exciting features which will tremendously simplify the database schemas and query statements. Based on SQL 1999/2003 object extensions, we designed a draft schema of object-relational design of CCDB. The UML class diagram is shown as Figure 2, and we also created the ODL-style schema.

```
interface Account: Object
    ( key: number )
{
    attribute string number;
    attribute numeric balance;
    attribute numeric limit;
    relationship Set<Person> authorizerdUsers inverse Person::authorizedAccounts;
}

class PerAccount: Account
{
    relationship Person owner inverse Person::accounts;
}

class OrgAccount: Account
{
    relationship Organization owner inverse Organization:accounts;
}

class Organization
    ( key: id )
{
    attribute string id;
    attribute string name;
```

```
    attribute string address;
    relationship SET<Account> accounts inverse OrgAccount::owner;
    relationship SET<Person> signers inverse Person::id;
}

class Person
    ( key: id )
{
    attribute string id;
    attribute string name;
    attribute string address;
    attribute date dob;
    relationship SET<Account> accounts inverse PerAccount::owner;
    relationship SET<Organization> signOrgs inverse Orgnization::id;
    relationship SET<Account> authorizedAccounts inverse Account::authorizedUsers;
}
```

With the power of object referencing, we can store all references into set-value types thus we do not need to create many relationship table/class. But unfortunately, PostgreSQL does not support all of the object-oriented features. In fact, it is very limited. Hence we have to adjust our design to cater to PostgreSQL.

1.2.1 Account Type Hierarchy

We have briefly talked about the split of account type in section 1.1.1. Now we describe it in detail. The first problem we encountered is that the owner of a credit card account can be either a person or an organization. Although the sample test data has different ids between Person and Organization, we can not assume all data satifsies this condition. If we add a new enum column, which labeling the owner type, into Account, then we can distingish it by $(ownerId, ownerType)$. But then we have another problem: we can not define a proper foreign key constraint under ownerId column.

Therefore, we decided to split the accounts into two types: Personal Account and Organizational Account. All the attributes of Account type are applicable to subtype entities. We defined this IsA relationship that relates Account, PerAccount and OrgAccount. The role Sub refers to PerAccount and OrgAccount, while the role Super refers to Account. In the view of object-oriented, Account is a super class while PerAccount and OrgAccount are subclasses, as shown in Figure 3. The subclasses inherit the features of the super class.

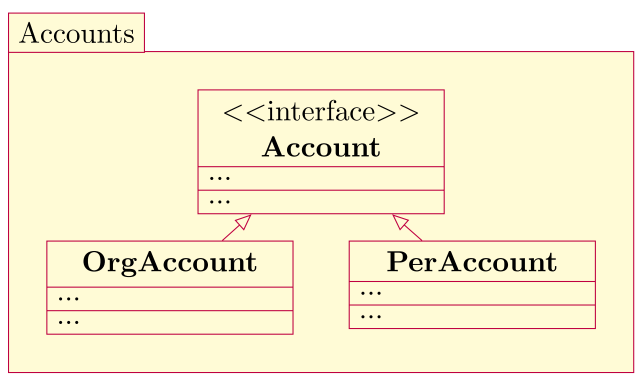


Figure 3: Hierarchy of Account Type

In addition, our partition on Account is disjoint. A credit card account can not be a personal account and an organizational account at the same time. Thus we will not have duplicate tuples when inserting data into non-disjoint subtypes in PostgreSQL (we will discuss it later in section). And the partition also satisfies the covering constraint since the union of the sets of instances PerAccount and OrgAccount equals the set of instances of Account.

We noticed that Person type and Organization type have many common attributes like id, name, address and they can both be the owner of a credit card account. A supertype, let us call it Owner, is not made for Person and Organization not only because we think those two types are essentially different but also they can not satisfy the covering constraint. If we create the Owner type for the convenience of defining the foreign key constraint of Account, then we can find some instances like 'p2- John' in Person doesn't own any card.

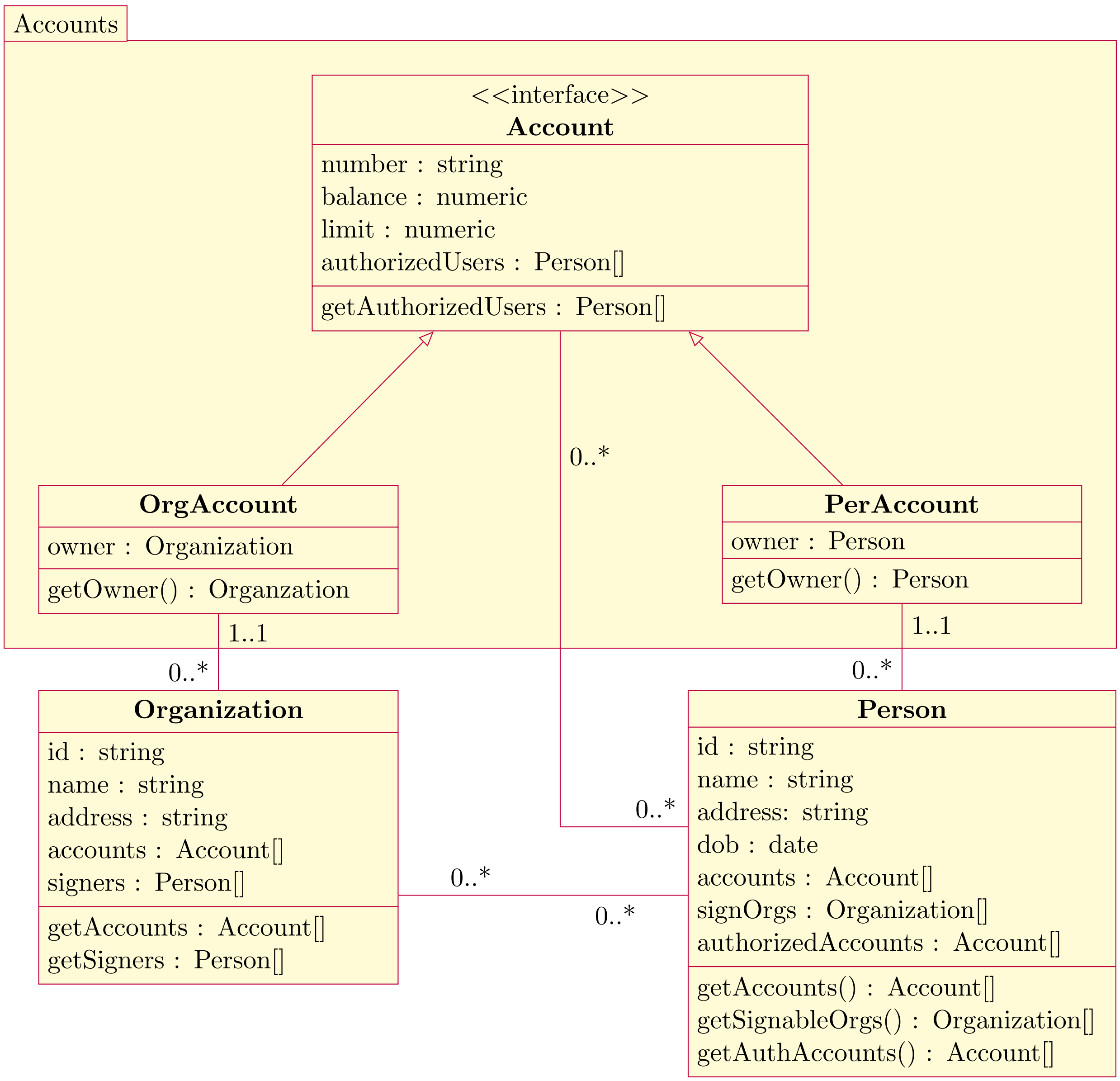


Figure 2: UML diagram for CCDB

1.2.2 Person Type Integrity

In our CCDB, a person can partcipate in many relationship. For example, a person can sign for an organization, can be authorized to use a card, and can also own some credit cards. Naturally, we can divide Person to Signer, AuthorizedUser and so on although the partition is not disjoint. At the begining we accept the Person Type Hierarchy and it seemed to have no big problem. But soon when we used PostgreSQL to insert data into tables, the trouble rose up.

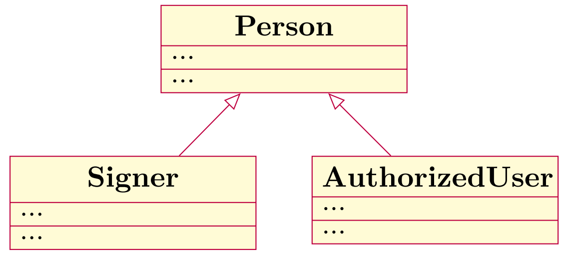


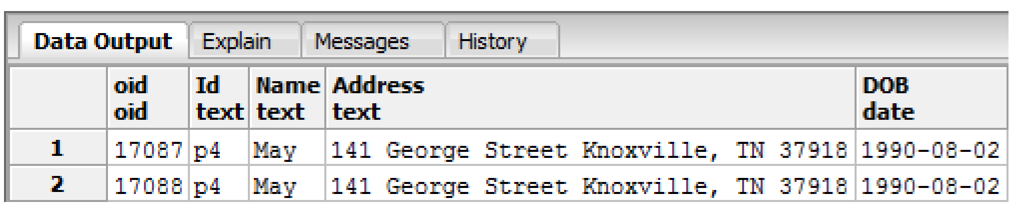
Figure 4: Early partition on Person

We used the concept of inheritance from object-oriented databases in PostgreSQL. Two tables, Signer and AuthorizedUser, inherit from Person. If we insert data into supertable Person, the subtables will have no data. That's understandable since the subtable has more attributes and constraints so that it can not extract data directly from supertable. But one person can be both a signer and an authorized user. If we insert data into subtables, we will have duplicate tuples in the view of supertable. For example 'p4 - May' is a signer of organization 'Acme', and also an authorized user of credit card c10. We insert the data of May into Signer and AuthorizedUser then we have two "identical" tuples in supertable Person.

We quote "identical" because they are actually different tuples in PostgreSQL database. In PostgreSQL documents, the behavior is described as *"INSERT always inserts into exactly the table specified."* Therefore, we have two different tuples with same values in table Signer and AuthorizedUser. When we run SELECT query on Person, PostgreSQL returns all data from tables inherited from Person.

To show that, we modified the table to explicitly store OIDs. Then we ran the following SQL command and got the result as shown in Figure 5.

```
SELECT oid, * FROM "Person";
```



Data Output | Explain | Messages | History

	oid oid	Id text	Name text	Address text	DOB date
1	17087	p4	May	141 George Street Knoxville, TN 37918	1990-08-02
2	17088	p4	May	141 George Street Knoxville, TN 37918	1990-08-02

Figure 5: Query on Person

The tuples were not identical because they came from different tables and they had different OIDs. The same-value tuples did not violate the primary key constraint because table Person was empty. We can verify that by add ONLY keyword to the query and the query will return 0 result.

```
SELECT oid, * FROM ONLY "Person";
```

In fact, that is not the only problem that inheritance of PostgreSQL rises. When we create a foreign key that points to a supertable, we will encounter exception saying that foreign key does not exist. It is because the supertable is usually empty and PostgreSQL does not extract data from subtables when handle foreign key constraints. This type of behavior has been described in the document:

All check constraints and not-null constraints on a parent table are automatically inherited by its children. Other types of constraints (unique, primary key, and foreign key constraints) are not inherited.

As a result, we need to handle the foreign key constraints which point to a supertable very carefully. The best way is to avoid using that. But we can also create some triggers to ensure the data consistency. Based on the complexity of implementation and the performance on PostgreSQL, we decided not to partition the Person type.

1.2.3 References in PostgreSQL

#What is in SQL, what is difference in PostgreSQL

#Oid, foreign key

#Does not support ARRAY ELEMENT foreign key referencing

#Use ARRAY + TRIGGER .vs. Use extra table + foreign keys

#About Account supertable referencing (TRIGGER)

1.2.4 CCDB Database on PostgreSQL

Using User-Defined Types in PostgreSQL One of the important object-oriented features is user-defined types. PostgreSQL has UDTs but does not support the inheritance of UDTs. Instead PostgreSQL provides table inheritance.

We also create Authorizes table for the relation Authorizes. The tricky part is the foreign key is pointing to a supertable, which rises trouble in PostgreSQL. In PostgreSQL, we create triggers to ensure the consistency. One trigger executes before insert or update on Authorizes. The other one executes after update or delete on Account. We will talk about triggers later in Object-oriented design of CCDB in section ?.?.?.

Table Inheritance in PostgreSQL

Functions

Triggers