CSE532 Project Report

Credit Card Database
via the Object-Oriented Extension of SQL

Author Name, #SBU ID

November 9, 2015

1 Database Design

Something about design, general

1.1 Entity-Relationship Design

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

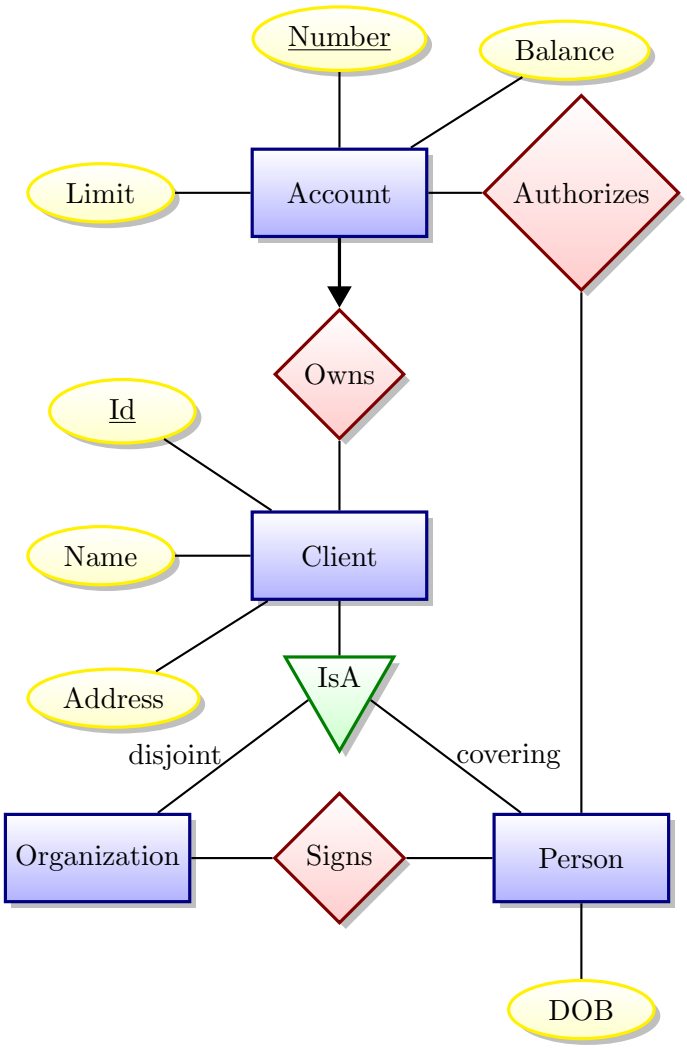

Figure 1: E-R diagram for CCDB

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 create a supertype Client. Every entity of type Organization and Person are subtypes of Client. This partition on Client satifies the disjointness constraint and covering constraint.

This supertype make it easy for us to define a proper foreign key constraint in Account table. The relationship "Owns"does not need extra table to store these relationships. In fact, every account can have exactly one owner, no more and no less. We draw an arrow pointing in the direction of the relationship's diamond in thick lines. We know that account Number is a key of Account and Owns. Thus, we can merge the attributes of Owns into Account. It is guaranteed that each Account tuple has exactly one corresponding Owns tuple, so no redundancy is created. Then we can concatenate Account and Owns tuples which have same account number to table AccountMergeWithOwns. For simplicity, we just call it Account.

The relationship Signs and Authorizes are many-to-many relationships. We have to create tables to store the relations, 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 client type hierarchy, we represent the IsA relationship in general representation. That means we create Client table stroing all information. And table Organization and Person both have a foreign key pointing to Client. The relationship Owns has been merged into Account. The SQL CREATE statements are as follows.

```
CREATE TABLE Account (
    Number  CHAR(20),
    Balance DECIMAL,
    Limit   DECIMAL,
    Owner   CHAR(20) NOT NULL,
    PRIMARY KEY (Number)
    FOREIGN KEY (Owner) REFERENCES Client(Id) )

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

CREATE TABLE Organization (
    Id      CHAR(20),
    PRIMARY KEY (Id),
    FOREIGN KEY (Id) REFERENCES Client(Id) )

CREATE TABLE Person (
    Id      CHAR(20),
    Dob     DATE,
    PRIMARY KEY (Id),
    FOREIGN KEY (Id) REFERENCES Client(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.

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

class Client: Object
{
    attribute string id;
    attribute string name;
    attribute string address;
    relationship SET<Account> accounts inverse Account::owner;
}

class Organization: Client
    ( key: id )
{
    relationship SET<Person> signers inverse Person::id;
}

class Person: Client
    ( key: id )
{
    attribute date dob;
    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.

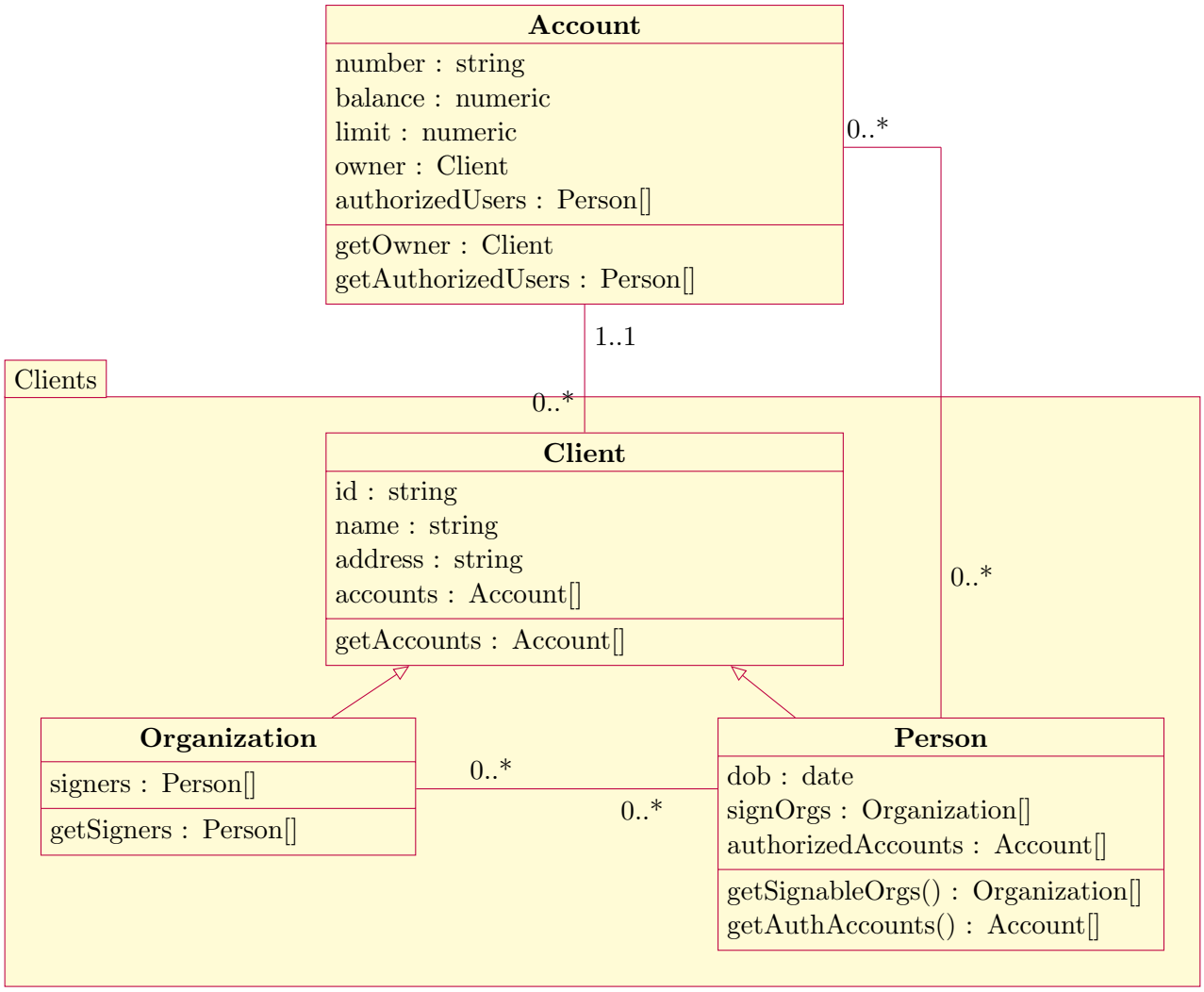

Figure 2: UML diagram for CCDB

1.2.1 Client Type Hierarchy

We have briefly talked about the supertype Client 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 create a supertype Client. All the attributes of Client type are applicable to subtype entities. We defined this IsA relationship that relates Client, Organization and Person. The role Sub refers to Organization and Person, while the role Super refers to Client. In the view of object-oriented, Client is a super class while Organization and Person are subclasses, as shown in Figure 3. The subclasses inherit the features of the super class.

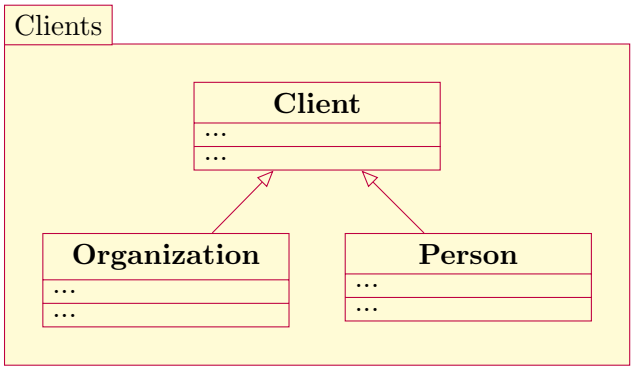

Figure 3: Hierarchy of Client Type

In addition, our partition on Client is disjoint. A client can not be a person and an organization at the same time. Thus we will not have duplicate tuples in subtables 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 Organization and Person equals the set of instances of Client.

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. Using the inheritance of object-oriented view, we can define those attributes once and reuse it in subclasses.

1.2.2 Person Type

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.

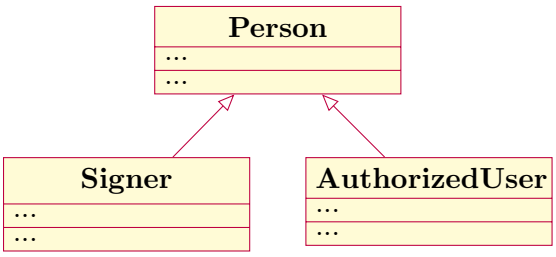

Figure 4: Rejected partition on Person

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";
```

The partition is not disjoint thus brings redundancy. We do not split the Person table in our design.

1.2.3 References in PostgreSQL

In SQL 1999/2003 object extensions, we can use object references to access related objects in a way similar to object-oriented language, like Java. In PostgreSQL, we can define a column type as an existing table. For example, we can define "owner" column as Person type in table PerAccount. But it is actually store a Person tuple in the column, not a real reference. PostgreSQL does not provide reference types. Instead, PostgreSQL has a kind of object identifier types. For safety issues, PostgreSQL only uses OIDs as primary keys for various system tables. User-created tables are not recommended to use OIDs.

Using OIDs in our tables does not simplify the query in our system. No function or call is provided to convert OID to object. If we want to get the object the OID represents, we have to do join query. Basically, it has nothing different from using our own foreign keys.

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. We can create some triggers to ensure the data consistency. Or we can insert duplicate tuples to both supertable and subtable. In our design, we have a foreign key pointing to supertable Client from table Account. We keep a same copy of every tuple of subtables Organization and Person in supertable Client.

In addition, PostgreSQL does not support array element foreign key referencing. In our original object-relational design, we have some array attributes in our classes. PostgreSQL can not define foreign key references on array. At first we want to use CHECK constraint, but those CHECK constraints in PostgreSQL does not support nested query.

So we have to use triggers to ensure the references if we want to use arrays in our tables. Or we can use extra relationship tables and foreign keys in an old way. It is ARRAY + TRIGGERS vs TABLE + FOREIGN KEY. We choose the latter in our final design and we have three reasons:

1. Using so many triggers is inefficient. For every many-to-many relationship, if we use arrays, we have to create triggers on every foreign key constraint. The triggers are for the action taken after update and delete. If we want to ensure participation constraints, we have to create triggers before insert. We do not want to write so many triggers. Using build-in foreign key constraints of PostgreSQL is fine with us.

2. Space efficiencies are asymptotically equal. Stroing the reference ids in array or table takes exactly the same space. Even if we compare the space taken by managing array structure or by managing new table, those two ways have same space complexities.

3. Class methods are global in PostgreSQL. We define some methods that return required tuples in our original design before. PostgreSQL stores all functions under schema, not table. That means if we move the array references out of table into a new relationship table, all functions in PostgreSQL have same accessibility.

So we decide to use extra relationship tables and foreign key references to store all many-to-many relationships. We also have some methods under the schema to provide us some object-oriented features to simplify our query.

1.2.4 CCDB Database on PostgreSQL

PostgreSQL has some object-relational database features. But many of them are different from what we think about. In this section we cretae our CCDB system on PostgreSQL. Before we go into details, we create a new UML class diagram for CCDB under PostgreSQL, as shown in Figure 6.

We remove the package Clients. Also, we omit all 0..* labels on association lines. Finally, all functions are grouped in a yellow box, which indicates the schema function library.

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. In our database, two UDTs are defined as accountType and clientType. The SQL commands are as follows:

```
CREATE TYPE public.accounttype AS
   ("number"  text,
    "balance" numeric,
    "limit"   numeric,
    "owner"   text);

CREATE TYPE public.clienttype AS
   ("id"     text,
    "name"    text,
    "address" text);
```

Table Inheritance in PostgreSQL PostgreSQL provides table inheritance. Based on the types and inheritance relationship, four tables are created as Account, Client, Organization and Person. We also create two relationship tables signs and authorizes. We include all primary key constraints and foreign key constraints as many as possible here.

```
CREATE TABLE public."Account"
OF accounttype
(
  "number" WITH OPTIONS  NOT NULL,
-- Inherited from type accounttype: balance,
```

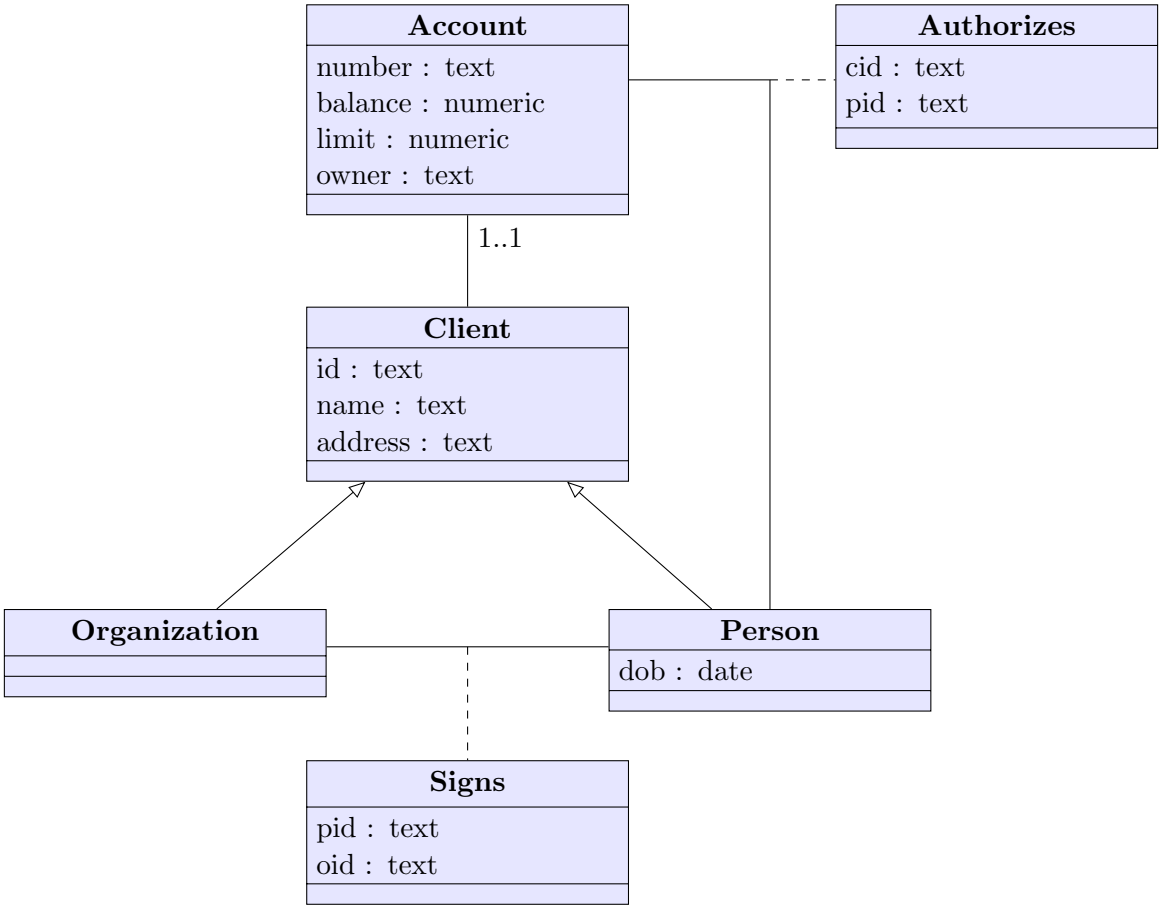

Figure 6: New UML diagram for CCDB

```
-- Inherited from type accounttype: "limit",
  owner WITH OPTIONS  NOT NULL,
  CONSTRAINT account_pkey PRIMARY KEY (number),
  CONSTRAINT "Account_owner_fkey" FOREIGN KEY (owner)
      REFERENCES public."Client" (id) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE public."Client"
OF clienttype
(
  id WITH OPTIONS  NOT NULL,
-- Inherited from type clienttype: name,
-- Inherited from type clienttype: address,
  CONSTRAINT "Client_pkey" PRIMARY KEY (id)
);

CREATE TABLE public."Organization"
(
-- Inherited from table "Client":  id text NOT NULL,
-- Inherited from table "Client":  name text,
-- Inherited from table "Client":  address text,
  CONSTRAINT "Organization_pkey" PRIMARY KEY (id)
)
INHERITS (public."Client");

CREATE TABLE public."Person"
(
-- Inherited from table "Client":  id text NOT NULL,
-- Inherited from table "Client":  name text,
-- Inherited from table "Client":  address text,
  dob date,
  CONSTRAINT "Person_pkey" PRIMARY KEY (id)
)
INHERITS (public."Client");

CREATE TABLE public.signs
(
  pid text NOT NULL,
  oid text NOT NULL,
  CONSTRAINT signs_pkey PRIMARY KEY (pid, oid),
  CONSTRAINT signs_oid_fkey FOREIGN KEY (oid)
      REFERENCES public."Organization" (id) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT signs_pid_fkey FOREIGN KEY (pid)
      REFERENCES public."Person" (id) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE public.authorizes
(
```

```
  cid text NOT NULL,
  pid text NOT NULL,
  CONSTRAINT authorizes_pkey PRIMARY KEY (cid, pid),
  CONSTRAINT authorizes_cid_fkey FOREIGN KEY (cid)
      REFERENCES public."Account" ("number") MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT authorizes_pid_fkey FOREIGN KEY (pid)
      REFERENCES public."Person" (id) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE CASCADE
);
```

Triggers We talked about we need triggers to ensure the constraints pointing to a supertable in section 1.2.3. Basically, we need to make supertable always has the tuples of its subtables. So we create triggers on Organization and Person. When the modify is done, do the same thing with supertable Client.

```
CREATE OR REPLACE FUNCTION public.tf_i_client()
  RETURNS trigger AS
$BODY$
  BEGIN
    INSERT INTO "Client" SELECT NEW.id, NEW.name, NEW.address;
    RETURN NEW;
  END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER tf_i_organization
  AFTER INSERT
  ON public."Organization"
  FOR EACH ROW
  EXECUTE PROCEDURE public.tf_i_client();

 CREATE TRIGGER tf_i_person
  AFTER INSERT
  ON public."Person"
  FOR EACH ROW
  EXECUTE PROCEDURE public.tf_i_client();

CREATE OR REPLACE FUNCTION public.tf_u_client()
  RETURNS trigger AS
$BODY$
  BEGIN
    UPDATE ONLY "Client" SET id=NEW.id, name=NEW.name, address=NEW.address WHERE id=OLD.id;
    RETURN NEW;
  END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER tf_u_organization
  AFTER UPDATE
  ON public."Organization"
  FOR EACH ROW
```

```
  EXECUTE PROCEDURE public.tf_u_client();

 CREATE TRIGGER tf_u_person
  AFTER UPDATE
  ON public."Person"
  FOR EACH ROW
  EXECUTE PROCEDURE public.tf_u_client();

CREATE OR REPLACE FUNCTION public.tf_d_client()
  RETURNS trigger AS
$BODY$
  BEGIN
    DELETE FROM "Client" WHERE id=OLD.id;
    RETURN NEW;
  END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER tf_d_organization
  AFTER DELETE
  ON public."Organization"
  FOR EACH ROW
  EXECUTE PROCEDURE public.tf_d_client();

 CREATE TRIGGER tf_d_person
  AFTER DELETE
  ON public."Person"
  FOR EACH ROW
  EXECUTE PROCEDURE public.tf_d_client();
```

Functions PostgreSQL allows function overloading. We create several basic functions to satifsy our need. For the required query, we also create some helper functions. We will talk about that in section 2.

```
CREATE OR REPLACE FUNCTION public."getAuthUsers"("Account")
  RETURNS SETOF "Person" AS
' SELECT p.* FROM "Person" p, authorizes a WHERE $1.number = a.cid AND a.pid = p.id '
  LANGUAGE sql;

CREATE OR REPLACE FUNCTION public."getOrgOwner"("Account")
  RETURNS SETOF "Organization" AS
' SELECT * FROM "Organization" o WHERE o.id = $1.owner '
  LANGUAGE sql;

CREATE OR REPLACE FUNCTION public."getPerOwner"("Account")
  RETURNS SETOF "Person" AS
' SELECT * FROM "Person" p WHERE p.id = $1.owner '
  LANGUAGE sql;

CREATE OR REPLACE FUNCTION public."getAccts"("Organization")
  RETURNS SETOF "Account" AS
' SELECT * FROM "Account" a WHERE a.owner = $1.id '
```

```
  LANGUAGE sql;

CREATE OR REPLACE FUNCTION public."getAccts"("Person")
  RETURNS SETOF "Account" AS
' SELECT * FROM "Account" a WHERE a.owner = $1.id '
  LANGUAGE sql;

CREATE OR REPLACE FUNCTION public."getSigners"("Organization")
  RETURNS SETOF "Person" AS
' SELECT a.* FROM "Person" a, signs s WHERE $1.id = s.oid AND a.id = s.pid '
  LANGUAGE sql;

CREATE OR REPLACE FUNCTION public."getSignOrgs"("Person")
  RETURNS SETOF "Organization" AS
' SELECT o.* FROM "Organization" o, signs s WHERE $1.id = s.pid AND o.id = s.oid '
  LANGUAGE sql;

CREATE OR REPLACE FUNCTION public."getAuthAccts"("Person")
  RETURNS SETOF "Account" AS
' SELECT a.* FROM "Account" a, authorizes s WHERE $1.id = s.pid AND s.cid = a.number '
  LANGUAGE sql;
```

Indexes Query Optimizer can take advantage of indexes to produce better execution plans. We want to speed up our queries and improve the performance of our Web applications. Although we can create indexes for every column used in our queries, PostgreSQL would waste space and time to determine which indexes to use. Meanwhile, extra cost will be added into operations like inserts, updates and deletes since indexes need to be updated.

To optimize the functions and the queries, we created 7 indexes in our database. We created an index on owner attribute of Account table since it is frequently used to join with Client, Organization and Person tables. We also considered to create an index on number attribute of Account table, but the effect was very limited and caused addition time on query 2. The id attributes of Organization and Person tables are also indexed. And finally, all attributes of authorizes and signs are indexed.

```
CREATE INDEX account_owner ON "Account" USING btree (owner);
CREATE INDEX organization_id ON "Organization" USING btree (id);
CREATE INDEX person_id ON "Person" USING btree (id);
CREATE INDEX signs_pid ON "signs" USING btree (pid);
CREATE INDEX signs_oid ON "signs" USING btree (oid);
CREATE INDEX authorizes_cid ON "authorizes" USING btree (cid);
CREATE INDEX authorizes_pid ON "authorizes" USING btree (pid);
```

2 Queries and Optimizations

The queries are the same as Project 1. PostgreSQL has not support universal quantification, so we convert query 3 into double-negation form. As for query 4 and 5, we created recursive queries using common table expressions.

To test the effect of optimization, we inserted 100,000 random tuples into Account, Organization and Person tables resepectively. Table authorizes and signs are also inserted 10,000 random tuples separately.

2.1 Query 1

Find all pairs of the form (user,signer), where user is an authorized user of an organization's credit card, signer is a person at that organization with signature authority, and the balance on the card is within $1,000 of the credit limit. For each user/signer, show Id and Name.

In this query, we treat "authorized user" as non-owner-and-signer authorized user to fit the test data sample output.

Supporting functions We already have all the supporting functions defined. What we need are getAuthUsers(Account), getOOwner(Account) and getSigners(Organization) in section 1.2.4.

Query SQL statement Using functions we talked above, we can build our query statement succinctly.

```
SELECT au.id, au.name, s.id, s.name
FROM "Account" oa, "getAuthUsers"(oa) au, "getOrgOwner"(oa) oo, "getSigners"(oo) s
WHERE oa.balance - oa.limit <= 1000 AND oa.balance - oa.limit >= -1000;
```

Optimization Table 1 shows the join fields of Query 1.

Table 1: Join table of Query 1

Function	*Join*			
getAuthUsers	1	Account.number	-	authorizes.cid
	2	authorizeds.pid	-	Person.id
getOrgOwner	3	Account.owner	-	Organization.id
getSigners	4	Organization.id	-	signs.oid
	5	signs.pid	-	Person.id

Although there are 5 joins in Query 1, the optimization effect of indexes is not obvious. This may be we haven't insert enough data into tables. Or the where clause of Query 1 reduces many tuples at the beginning. The query runtimes before and after optimization are all 1.3 seconds.

2.2 Query 2

Find all users (Id, Name) who own four or more cards and are authorized non-owner users for three or more other cards.

Supporting functions We defined two supporting functions to this query: getAcctNum(Person) and getAuthAcctNum(Person).

```
CREATE OR REPLACE FUNCTION public."getAcctsNum"("Person")
  RETURNS bigint AS
' SELECT count(*) FROM "getAccts"($1)'
  LANGUAGE sql;

CREATE OR REPLACE FUNCTION public."getAuthAcctsNum"("Person")
  RETURNS bigint AS
' SELECT count(*) FROM "getAuthAccts"($1) '
  LANGUAGE sql;
```

Query SQL statement

```
SELECT p.id, p.name
FROM "Person" p
WHERE "getAcctsNum"(p)>=4 AND "getAuthAcctsNum"(p)>=3;
```

Optimization Table 2 shows the join fields of Query 2.

Table 2: Join table of Query 2

Function	*Join*			
getAcctsNum	1	Person.id	-	Account.owner
getAuthAcctsNum	2	Person.id	-	authorizes.pid
	3	authorizes.cid	-	Account.number

The optimization effect in Query 2 is very significant. The runtime before optimization is 18 minutes 47 seconds while 5.6 seconds is the runtime after optimization.

2.3 Query 3

Find the credit cards (acct. numbers) all of whose signers (i.e., people with signature authority of the organizations that own those cards) also own personal credit cards with credit limits at least $25,000.

Supporting functions We defined one supporting function getMaxLimit(Person) to this query.

```
CREATE OR REPLACE FUNCTION public."getMaxLimit"("Person")
  RETURNS numeric AS
' SELECT max(a.limit) FROM "getAccts"($1) a '
  LANGUAGE sql;
```

Indexes

```
CREATE INDEX organization_id_index
  ON "Organization"
  USING btree
  (id COLLATE pg_catalog."default");

CREATE INDEX person_id_key
  ON "Person"
  USING btree
  (id COLLATE pg_catalog."default");
```

Query SQL statement Since PostgreSQL does not support universal quantification, we have to use double-negation to create equivalent query. We convert “for all ... at least” to “not exist ... less than”.

```
SELECT DISTINCT a.number
FROM "Account" a, "getOrgOwner"(a) oo, "getSigners"(oo) s
WHERE NOT "getMaxLimit"(s) < 25000;
```

Optimization Table 3 shows the join fields of Query 3.

The optimization effect in Query 3 is not obvious. The runtime before optimization is 1012 milliseconds while 845 milliseconds is the runtime after optimization.

Table 3: Join table of Query 3

Function	*Join*			
getOrgOwner	1	Account.owner	-	Organization.id
getSigners	2	Organization.id	-	signs.oid
	3	signs.pid	-	Person.id
getMaxLimit	4	Account.owner	-	Person.id

2.4 Query 4

Find all pairs (U,C) where U is an indirect user of the credit card C. (For each user, show Id and Name. For credit cards, show the account number.)

Supporting functions We need to define a new function that returns proper authorized users. Basically it is getAuthUsers function plus getSigners if the card owner is an organization.

```
CREATE OR REPLACE FUNCTION public."getNonOwnerAuthUsers"("Account")
  RETURNS SETOF "Person" AS
$BODY$
  SELECT p.*
  FROM "Person" p, authorizes a
  WHERE $1.number = a.cid AND a.pid = p.id
  UNION
  -- union signer if owner is an organization
  SELECT p.*
  FROM "Person" p, signs s
  WHERE $1.owner = s.oid AND s.pid = p.id
$BODY$
  LANGUAGE sql;
```

Query SQL statement WITH statement (CTE) is used in this query to do recursive query.

```
WITH RECURSIVE indirect_user(id, name, number) AS (
    SELECT aa.id, aa.name, a.number
    FROM "Account" a, "getNonOwnerAuthUsers"(a) aa
  UNION
    SELECT aa.id, aa.name, iu.number
    FROM indirect_user iu, "Account" a, "getNonOwnerAuthUsers"(a) aa
    WHERE iu.id = a.owner
)
SELECT *
FROM indirect_user;
```

Optimization Table 4 shows the join fields of Query 4.

The optimization effect in Query 4 is obvious. The runtime before optimization is 8 minutes and 4 seconds while 10.4 seconds is the runtime after optimization.

2.5 Query 5

Find the total of all balances for the credit cards that have Joe as one of the indirect users.

Supporting functions Same as Query 4.

Table 4: Join table of Query 4

Function	*Join*			
getNonOwnerAuthUsers	1	Account.number	-	authorizes.cid
	2	authorizes.pid	-	Person.id
	3	Account.owner	-	signs.oid
	4	signs.pid	-	Person.id
indirectUser	5	indirectUser.id	-	Account.owner

Query SQL statement WITH statement (CTE) is used in this query to do recursive query.

```
WITH RECURSIVE indirect_user(id, name, number) AS (
    SELECT aa.id, aa.name, a.number
    FROM "Account" a, "getNonOwnerAuthUsers"(a) aa
  UNION
    SELECT aa.id, aa.name, iu.number
    FROM indirect_user iu, "Account" a, "getNonOwnerAuthUsers"(a) aa
    WHERE iu.id = a.owner
)
SELECT sum(a.balance)
FROM indirect_user iu, "Account" a
WHERE iu.name = 'Joe' AND iu.number = a.number;
```

Optimization Table 5 shows the join fields of Query 5.

Table 5: Join table of Query 5

Function	*Join*			
getNonOwnerAuthUsers	1	Account.number	-	authorizes.cid
	2	authorizes.pid	-	Person.id
	3	Account.owner	-	signs.oid
	4	signs.pid	-	Person.id
indirectUser	5	indirectUser.id	-	Account.owner
SELECT	6	indirectUser.number	-	Account.number

The optimization effect in Query 5 is same as Query 4. The runtime before optimization is 8 minutes and 21 seconds while 10.5 seconds is the runtime after optimization.

3 CCDB System Interface

The interface to our CCDB system is realized with the front end JSP webpage and the back end servlet. A webpage provides an interface where users are able to invoke one of the five queries described in section 2. The servlet is responsible for interacting with the database and returning the query results to the browser.

3.1 Front End JSP

A JSP is a webpage combining the traditional HTML and Java code. In our webpage, five links are available for each query; clicking a link is viewed as sending a request to the servlet. Furthermore, calling the servlet is achieved using JavaScript.

```
<ul class="submenu">
    <li><a href="#" onclick="javascript: query(1);">Query 1</a></li>
```

```
    <li><a href="#" onclick="javascript: query(2);">Query 2</a></li>
    <li><a href="#" onclick="javascript: query(3);">Query 3</a></li>
    <li><a href="#" onclick="javascript: query(4);">Query 4</a></li>
    <li><a href="#" onclick="javascript: query(5);">Query 5</a></li>
</ul>

<script language="JavaScript">
    function query(id) {
        resulttable.innerHTML = "<iframe src=\"QueryProcessingServlet?Query=" + id + "\"
                                scrolling=\"no\"></iframe>";
    }
</script>
```

3.2 Back End Servlet

The servlet, which is indeed a Java class named QueryProcessingServlet, generates dynamic HTML content showing the query results. It receives the request from JSP, processes such request, and constructs a response sent back to the browser. Detailed servlet's tasks are described as follows.

3.2.1 Establish a Connection to the CCDB

With the use of JDBC driver, the servlet initiates a connection to the CCDB system.

```
// Load JDBC driver
Class.forName("JDBC.driver");

// Create a connection
// DatabaseURL: our CCDB url (e.g. "jdbc:postgresql://127.0.0.1:5432/ccdb")
Connection conn = DriverManager.getConnection( DatabaseURL, UserName, Password);
```

3.2.2 Create a SQL Statement and Obtain the Results

The servlet creates a SQL statement for sending a SQL query to the database and obtaining the produced results.

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE)

// sql is a String object containing one of our five SQL queries.
ResultSet rs = stmt.executeQuery(sql);
```

3.2.3 Return Query Results to the Browser

The servlet creates a PrintWriter object to compose a text-output stream. The stream integrates the query results into the HTML content so that the browser is able to render it.

```
HttpServletResponse response;
PrintWriter out = response.getWriter();

out.println( "<table>" );

// Print the attributes
out.println( "<thead> <TR>" );
for( int i = 1; i <= columnCnt; ++i )
```

```
{
    String label = rs.getMetaData().getColumnLabel( i );
    out.println( "<TD><b>"+ label +"</b></TD>" );
}
out.println( "</thead> </TR>" );

// Print the values
while (rs.next())
{
    out.println( "<TR align='left'>" );
    for( int i = 1; i <= columnCnt; ++i )
    {
        out.println( "<TD>"+ rs.getString(i) +"</TD>" );
    }
    out.println( "</TR>" );
}

out.println( "</table>" );
```

4 Results

A user first opens a browser and types the URL (`http://localhost:8080/ccdb_oo/index.jsp`) to connect to the server. Then, a webpage allowing the user to select a SQL query shows up. Once a query is invoked, a table consisting of query results will be displayed. Figure 7 shows a sample result in which the user invokes the first query.

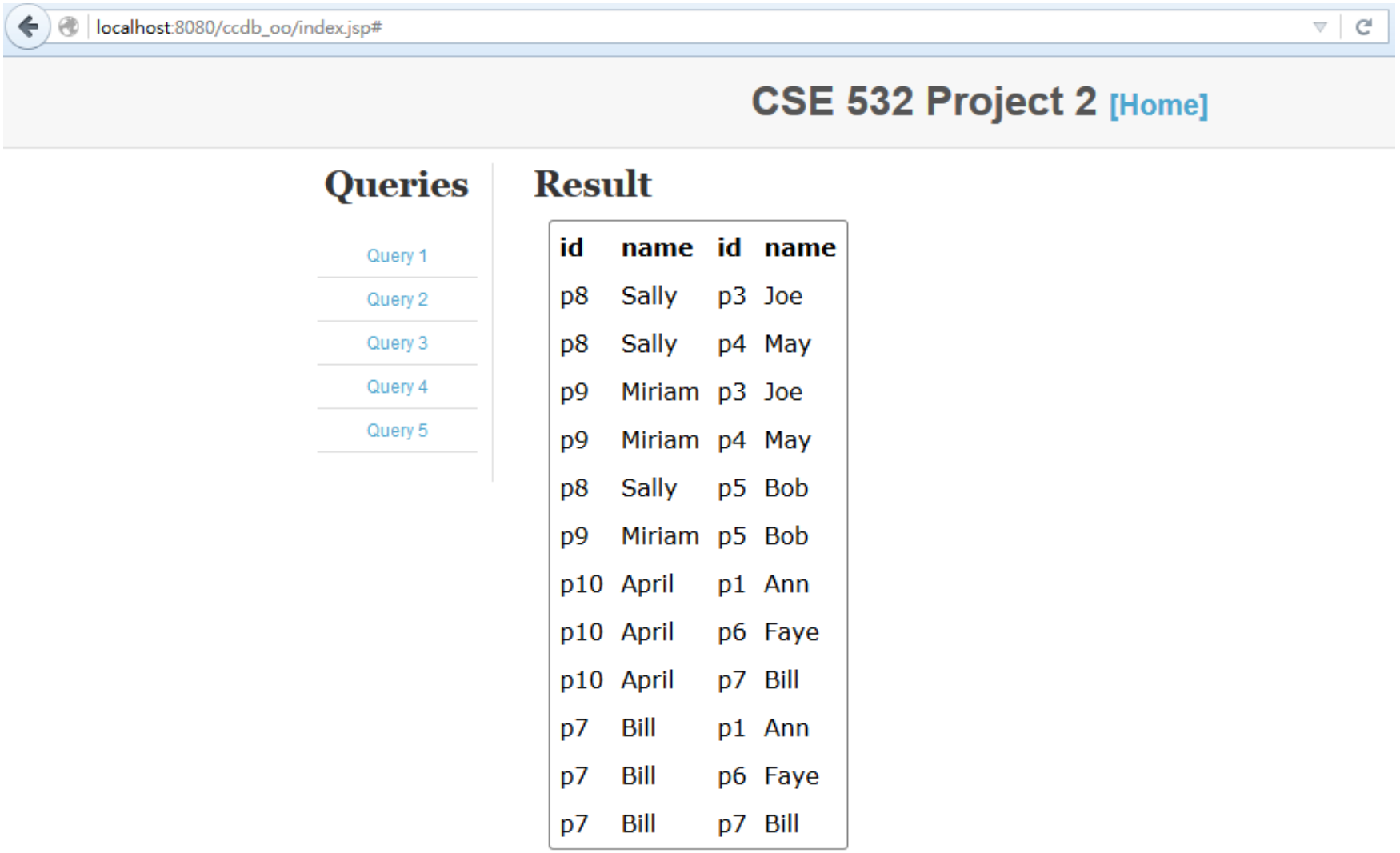

Figure 7: Result of the first query