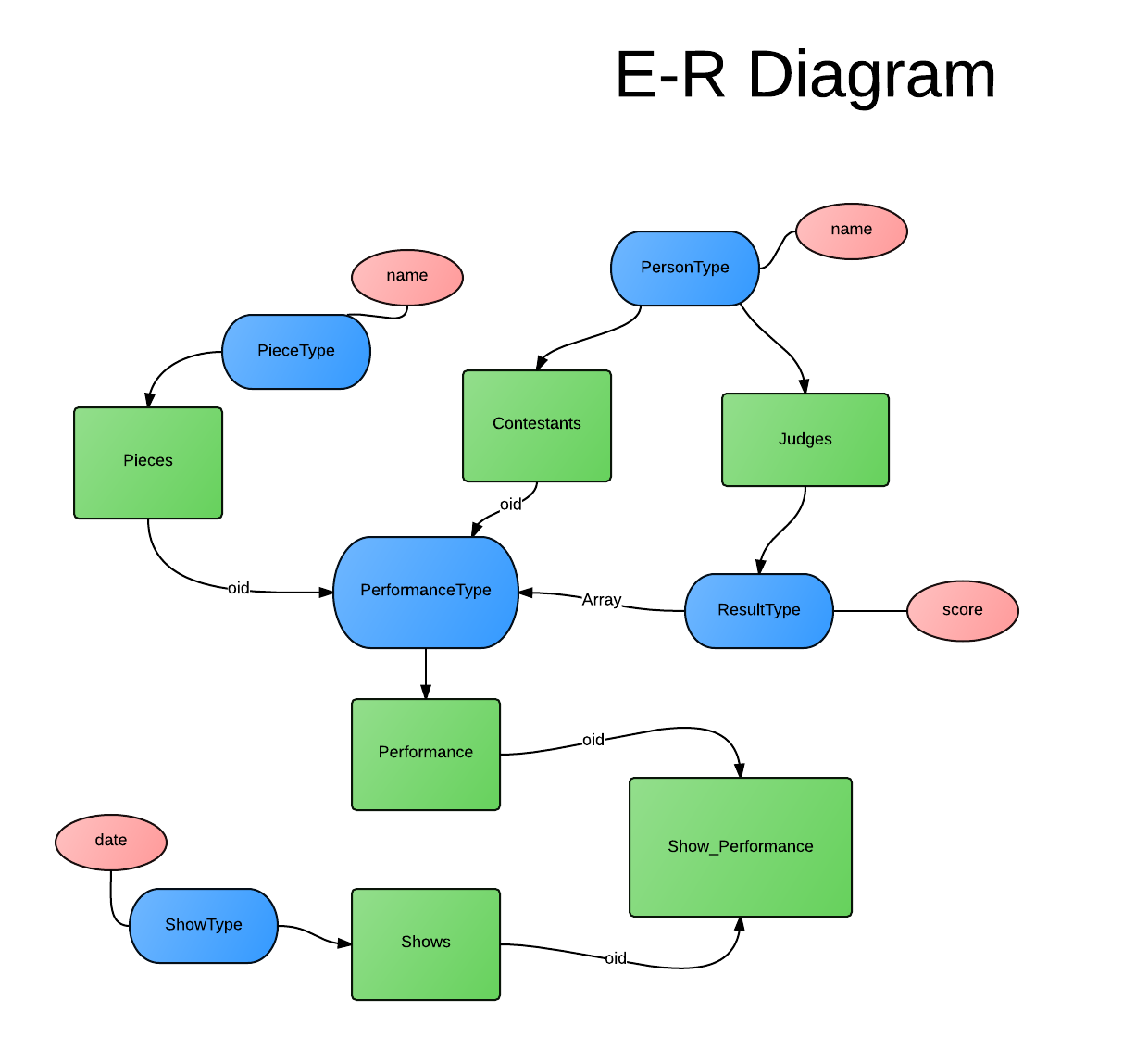
CSE 532 Project 2 Report

109892492 Hao-Tsung Yang

109247869 Kin Sum Liu

We, pledge our honor that all parts of this project were done by us alone and without collaboration with anybody else.

**Entity-Relationship Diagram**



Pinks are basic types, blues are UDTs, greens are tables.

**Database Scheme**

In this projects, we create 5 user-defined types: PERSON type, PIECE type, RESULT type, PERFORMANCE type and SHOW type

PersonType represents the person class. A person is identified by the name attribute uniquely which is of the basic type text.

PieceType represents the piece class which is identified by the name attribute uniquely.

ResultType represents the data type for the result object which involves two attributes: judge and score.This is an object that describes the result of a contestant on a piece in a performance.

PerformanceType represents the type for a performance. A performance describes the multiple results received by a contestant on a particular piece. So it contains an array of objects of ResultType in the attribute results. It makes the database more object-relational.

ShowType represents the type for a show which is uniquely identified by the date.

Also, create 5 typed tables: Contestants, Judges, Pieces, Shows, Performances

Contestants stores all the contestants of type PersonType.

Judges stores all the judges of type PersonType.

Pieces store all the pieces of type PieceType.

Performances store all the performances of type PerformanceType.

Shows store all the shows of type ShowType

Since the documentation indicates that “Typed table implement a subset of the SQL standard”. So the all typed tables do not contain the self-referencing column. We decide to use the OID feature to support the column explicitly. Therefore, all 5 typed tables are enabled with OID and the OID is the primary key for the objects (entries) in the tables.

Create a table called Show\_Performances that relates entries in Shows and Performances. The reason for that is explained in the integrity constraints section.

**Integrity Constraints**

According to the SQL standard, we should define the PerformanceType like :

CREATE TYPE PerformanceType AS (

contestant REF(PersonType) SCOPE Contestants

…

}

Since there is a difference between the SQL standard and the implementation in postgresql, we implement the referential integrity by using the foreign keys to reference the objects (entries) in the typed table. So in the creation of Performances table:

CREATE TABLE Performances OF PerformanceType (

Contestant WITH OPTIONS REFERENCES Contestants(oid),

Piece WITH OPTIONS REFERENCES Pieces(oid),

PRIMARY KEY (oid)

) WITH (OIDS);

The attribute Contestant and Piece for an object in Performances should reference the respective objects in the Contestants and Pieces

Another issue is that the current version of postgresql does not support the foreign key constraint on array member. So in order to relate a Show with multiple Performances.We use a relational approach to create a table Show\_Performances that relate and reference objects in Shows and Performances:

CREATE TABLE Show\_Performances (

show oid REFERENCES Shows(oid),

performance oid REFERENCES Performances(oid)

);

As the name of contestants, judges and pieces should be unique (it can be duplicate when we have more identifier for the objects like SSN), we have the UNIQUE constraints on them:

UNIQUE(name)

CHECK constraints are used on the names such that it cannot accept a contestant/judge/piece with empty string as the name:

name WITH OPTIONS CHECK (char\_length(name) > 0)

**SQL Commands: CREATE TYPE**

CREATE TYPE PersonType AS (

name text

);

CREATE TYPE PieceType AS (

name text

);

CREATE TYPE ResultType AS (

judge text,

score integer

);

CREATE TYPE PerformanceType AS (

Contestant oid,

Piece oid,

results ResultType[]

);

CREATE TYPE ShowType AS (

showdate date

);

**SQL Commands: CREATE TABLE**

CREATE TABLE Contestants OF PersonType (

UNIQUE(name),

PRIMARY KEY (oid),

name WITH OPTIONS CHECK (char\_length(name) > 0)

) WITH (OIDS);

CREATE TABLE Judges OF PersonType (

UNIQUE(name),

PRIMARY KEY (oid),

name WITH OPTIONS CHECK (char\_length(name) > 0)

) WITH (OIDS);

CREATE TABLE Pieces OF PieceType(

UNIQUE(name),

PRIMARY KEY (oid),

name WITH OPTIONS CHECK (char\_length(name) > 0)

) WITH (OIDS);

CREATE TABLE Performances OF PerformanceType (

Contestant WITH OPTIONS REFERENCES Contestants(oid),

Piece WITH OPTIONS REFERENCES Pieces(oid),

PRIMARY KEY (oid)

) WITH (OIDS);

CREATE TABLE Shows OF ShowType(

UNIQUE(showdate),

PRIMARY KEY (oid)

) WITH (OIDS);

CREATE TABLE Show\_Performances (

show oid REFERENCES Shows(oid),

performance oid REFERENCES Performances(oid)

);

**SQL Commands: CREATE VIEW**

The view Direct\_Coaudition is used in query 1 and 5.

CREATE VIEW Direct\_Coaudition AS (

SELECT C1.name AS name1, C2.name AS name2

FROM Contestants C1, Contestants C2,

Show\_Performances SP1, Show\_Performances SP2,

Performances P1, Performances P2

WHERE SP1.show = SP2.show AND // select same show of different performances

SP1.performance = P1.oid AND

SP2.performance = P2.oid AND

P1.piece = P2.piece AND //with same piece

P1.contestant = C1.oid AND

P2.contestant = C2.oid AND

C1.name != C2.name AND // but different contestants

P1.results && P2.results // with at least one same result from same judge

);

The recursive view Coaudition is used in query 5.

CREATE RECURSIVE VIEW Coaudition (name1, name2) AS (

SELECT \*

FROM Direct\_Coaudition //select all from Direct\_Coaudition first as base case

UNION

SELECT C1.name1,C2.name2

FROM Coaudition C1, Direct\_Coaudition C2 //build it recursively

WHERE C1.name2 = C2.name1 AND

C1.name1 != C2.name2

);

The view Shows3Judges is used in query 3.

CREATE VIEW Shows3Judges AS //find shows with 3 judges

SELECT oid, showdate // Select the show’s oid that

FROM Shows S

WHERE EXISTS( // exists some of the contestants

SELECT P.contestant

FROM Show\_Performances SP, Performances P

WHERE S.oid = SP.show AND

SP.performance = P.oid AND

array\_length(P.results, 1) > 2 // that have more than 2 results (means there are at least 3 judges)

);

**SQL Query: Query 1**

*Find all pairs of contestants who happened to audition the same piece during the same show and got the same score from at least one judge*

SELECT \* FROM Direct\_Coaudition WHERE name1 < name2; //show the tuples in Direct\_Coaudition without duplicated data

**SQL Query: Query 2**

*Find all pairs of contestants who happened to audition the same piece (in possibly different shows) and got the same average score for that piece.*

SELECT C1.name AS name1, C2.name AS name2

FROM Contestants C1, Contestants C2,

Performances P1, Performances P2

WHERE P1.piece = P2.piece AND // contestants perform same piece

P1.contestant = C1.oid AND

P2.contestant = C2.oid AND

C1.name < C2.name AND // they have different names

( SELECT avg(r.score) FROM unnest(P1.results) r) =

( SELECT avg(r.score) FROM unnest(P2.results) r) // have same average scores.

;

**SQL Query: Query 3**

*Find all pairs of contestants who auditioned the same piece in (possibly different) shows that had at least 3 judges and the two contestants got the same highest score.*

SELECT C1.name AS name1, C2.name AS name2

FROM Contestants C1, Contestants C2, //select 2 contestants

Shows3Judges S1, Shows3Judges S2,

Show\_Performances SP1, Show\_Performances SP2,

Performances P1, Performances P2

WHERE S1.oid = SP1.show AND // that perform in the show of show3judge

S2.oid = SP2.show AND

SP1.performance = P1.oid AND

SP2.performance = P2.oid AND

P1.piece = P2.piece AND //with same piece

P1.contestant = C1.oid AND

P2.contestant = C2.oid AND

C1.name < C2.name AND

( SELECT max(r.score) FROM unnest(P1.results) r) = ( SELECT max(r.score) FROM unnest(P2.results) r)

; // and same max score

**SQL Query: Query 4**

*Find all pairs of contestants such that the first contestants has performed all the pieces of the second contestant (possibly in different shows)*

SELECT C1.name AS name1, C2.name AS name2

FROM Contestants C1, Contestants C2

WHERE

AND NOT EXISTS( // and not exist

(

SELECT piece // a piece performed by C2

FROM Performances P

WHERE P.contestant = C2.oid

)

EXCEPT //but not performed by C1

(

SELECT piece

FROM Performances P

WHERE P.contestant = C1.oid

)

) AND

C1.name != C2.name

;

**SQL Query: Query 5**

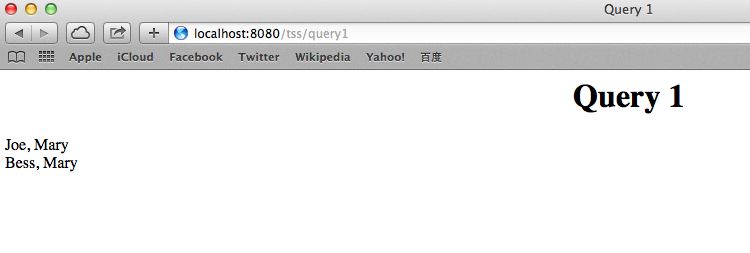
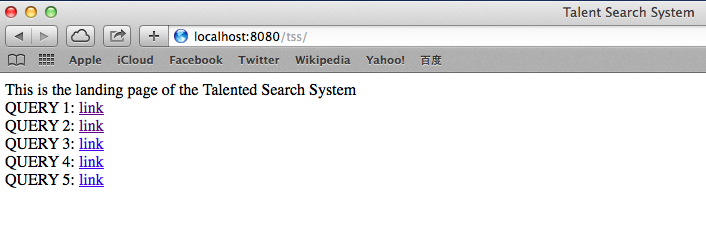
*Find all chained co-auditions. A chained co-auditions is the transitive closure of the following binary relation: X and Y (directly) co-auditioned iff they both performed the same piece in the same show and got the same score from at least one (same) judge. Thus, a chained co-audition can be either a direct or an indirect co-audition*

SELECT \* FROM Coaudition C WHERE C.name1<C.name2;

**Installation guide & Web Interface**

To have everything ready, you need to have postgresql ready. Create a database under the name tss, on port 5432. The user name and password should be kinsumliu and pw. Otherwise, my java code wont be able to connect to the database.

For web interface, we need Apache Tomcat, the postgresql JDBC driver, Eclipse EE and my source code which is an eclipse dynamic web project. The landing page looks like this with 5 links that execute the 5 SQL queries:



If you click one of these links, it directs to a page that displays the result:

**Teaming**

Division of labor