# README for DB_project1_part4

## Team Members

**Mingfei ge/ UNI: mg3534**
**Yuan Feng/ UNI: yf2338**

## File List

**EXPDAT.DMP**———————————Transcript of Schema: changed tables for PART 4
**queires.txt**———————————Three queries and explanations, based on new tables
**README.pdf**———————————This README file

## Oracle Account Name

**mg3534**

## Description for Schema Design

**-Objects added:**
1. STAFF
2. TA
3. PROF

**-Entities using the new types (tables):**
1. TA_TAB (TA)
2. PROF_TAB (PROF)

**-Relations using the new types (tables):**
1. ASSISTS (TA, COURSE)
2. TEACHES (PROF, COURSE)

**-Object relational features used & Explanation:**
1. Methods
2. Inheritance
3. References
4. TABLE
5. VALUE

**Explanation:**

**Method: MEMBER FUNCTION get_id() RETURN NUMBER**
is added to the super-type STAFF and so is inherited to the sub-types TA and PROF, which returns the object's STAFF_ID, which is the primary key in tables TA_TAB and PROF_TAB, that uniquely identify all STAFF objects.
With this method we can easily get the STAFF_ID for what every staff object of whatever the specific type, for convenience of referencing in relational schema.

**Method: MEMBER FUNCTION type_is() RETURN CHAR**
is added to the super-type STADD and so is inherited to the sub-types TA and PROF, but is overridden in both types, in different ways. In the super type, the method only returns "staff" which is the generic type. But in TA type, it returns "staff as TA" to describe the specificity of TA sub-type. And, in PROF type, it returns "staff as PROFESSOR" to describe the specificity of PROF sub-type.
If there is any table that can contain all objects in the generic type STAFF, regardless of their sub-type, we can use this overridden method to check the sub-types of the objects in the table.

**Inheritance:**
As described above, there is a super type STAFF which is generic and contains both the sub-types TA and PROF, i.e. both TA and professors are staffs in the school, which is semantically easy to be understood.
In the original relational-only schema, we have a table for storing all the staffs, which is not directly related with other entities, but only be referenced by the relation of TA & course and PROF and course. So, this is to some extend redundant. But in the relational-only schema, build separately the entities TA and professors will contain more redundancies which are the attributes they have in common. With object-relational schema. We build the generic type, and only use this to instance the sub-types and use sub-types to build tables. Types would not waste any storage, but tables do. So, with this schema, we eliminate out the STAFF table and save the storage of database.

**References:**
In table TA_TAB and PROF_TAB which store the TA and PROF objects, and in ASSISTS and TEACHES tables, which build relations between TA & COURSE and PROF & COURSE, we only store the references of TAs and PROFs in the relation table. Also, the reference is stored as the generic type staff, while with scope constraint of TA_TAB and PROF_TAB.
This feature guarantees that the same data is only stored for once in the database, i.e. those TA and PROF objects, and once we need to build relationship in the database based on them, we only use the references, like the pointers to them, which dramatically reduce the redundancies compared with relational-only schema. And, with DEREF as shown in the queries, we can easily reconstruct the object data anywhere. So, now, given a course_id, which is the identifier for courses, we can reconstruct all information of the professor teaching the course and the TAs assisting the teaching, without waste of any storage.

**TABLE & VALUE:**

The TA_TAB and PROF_TAB store row-objects, i.e. the table only store some specific type of object, we can see attributes of the object as different column in the table, or the whole object as a row in the table.
We directly use VALUE(o) to obtain the specific objects using queries, and with dot to specify attributes we want from the object, like shown in queries.