ACM SIGMOD Record, Volume 24, Number 3, September 1995, pp. 23 - 27.

Normalization in OODB Design

Byung S. Lee
Graduate Programs in Software
University of St. Thomas
St. Paul, Minnesota
bslee@stthomas.edu

Abstract

When we design an object-oriented database schema, we need to normalize object classes as we do for relations when designing a relational database schema. However, the normalization process for an object class cannot be the same as that of a relation, because of the distinct characteristics of an object-oriented data model such as complex attributes, collection data types, and the usage of object identifiers in place of relational key attributes. We need only one kind of dependency proposed here -- the object functional dependency -- which specifies the dependency of object attributes with respect to the object identifier. We also propose the object normal form of an object class, for which all determinants of object functional dependencies are object identifiers. There is no risk of update anomalies as long as all object classes are in the object normal form.

1. Introduction

Do we need normalization when designing an object-oriented database (OODB) schema? As pointed out by Pratt and Adamski in [1], "some proponents of the object-oriented approach claim that there is no need to normalize." This claim seems to be based on the fact that there is no notion of a key attribute in an object class. It is certainly not valid from a general database design perspective. We will run into the same problems as we see in a relational database if object classes are not normalized. In this regard, the following specific questions are addressed in this paper: (1) Are there risks of update anomalies in an object class? (2) If so, can they be eliminated by normalization? (3) How is the normalization different from that of a relation? (4) What will be the formal normalization process? (5) How can the normalization be performed in practical applications?

We will first review the normalization of relations in Section 2, and then present the normalization of object classes in Section 3. In an OODB design, we need only one kind of normal form -- *object normal form* -- which is the counterpart of the relational Boyce-Codd Normal Form (BCNF). Normalization to the object normal form is sufficient for rendering object classes free from the risk of update anomalies. We also need to re-define the functional dependency of a relational database to an *object functional dependency.* Object functional dependency specifies the dependency of object attributes on the object identifier. Examples of a practical normalization process will be shown for both a relation and an object class.

2. Normalization of Relations: Review

A poorly designed relation incurs the overhead of handling redundant data and the risk of causing update anomalies. The typical fix of the design is to decompose the relation into two or more relations with no such problems. In a formal method, we use the notion of a dependency such as a functional dependency (denoted by $X \rightarrow Y$ where X and Y are the sets of attributes) and a multi-valued dependency (denoted by $X \rightarrow\rightarrow Y$). Table 1 shows the constraints each normal form should preserve with respect to a dependency [3,4]. A normalization is the process of decomposing a relation into those that satisfy the normal form constraints. A fully normalized relation is in the BCNF when there are functional dependencies only, and in the fourth normal form when multi-valued dependencies exist as well. These two normal forms are sufficient in practical design cases. The fifth normal form exists in theory in consideration for join dependencies. However, it is of little practical usage because of the difficulty of identifying join dependencies. In short, a fully normalized relation is one whose only dependencies are functional dependencies that appear as shown in Fig. 1, that is, every non-key attribute is determined by the entire and only the key attribute and there is no multi-valued dependency

Normal form	Constraints
First normal form (1NF)	No composite attribute and no repeated (multi-valued) attribute are allowed. In other words, all attributes are atomic, i.e., simple and single-valued.
Second normal form (2NF)	In the 1NF and there exists no partial functional dependency on the key attribute. In other words, all attributes are dependent on the entire key.
Third normal form (3NF)[1] or BCNF	In the 2NF and there exists no transitive functional dependency. In other words, all determinants are key attributes, where a determinant refers to the left hand side of a functional dependency $X \rightarrow Y$.
Fourth normal form (4NF)	In the 3NF and there exists no multi-valued dependency.
Fifth normal form (5NF)	In the 4NF and all join dependencies are 'consequences of' [3] key attributes. In other words, each projection in a join dependency contains a key attribute.

Table 1. Normal forms and their constraints

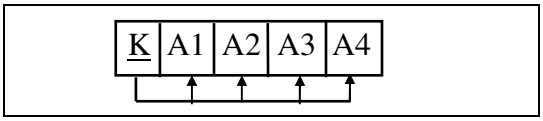


Fig. 1. A fully normalized relation

In real-world applications, database designers often rely on intuitions to decompose a relation into fully normalized relations. The most general case of a decomposition can be performed in the following three steps

i. Create a referenced relation if one does not exist.

[1] A 'new' third normal form [2]

ii. Introduce a foreign key if one does not exist.

iii. Move decomposed attributes to the referenced relation. Rename the attributes if necessary.

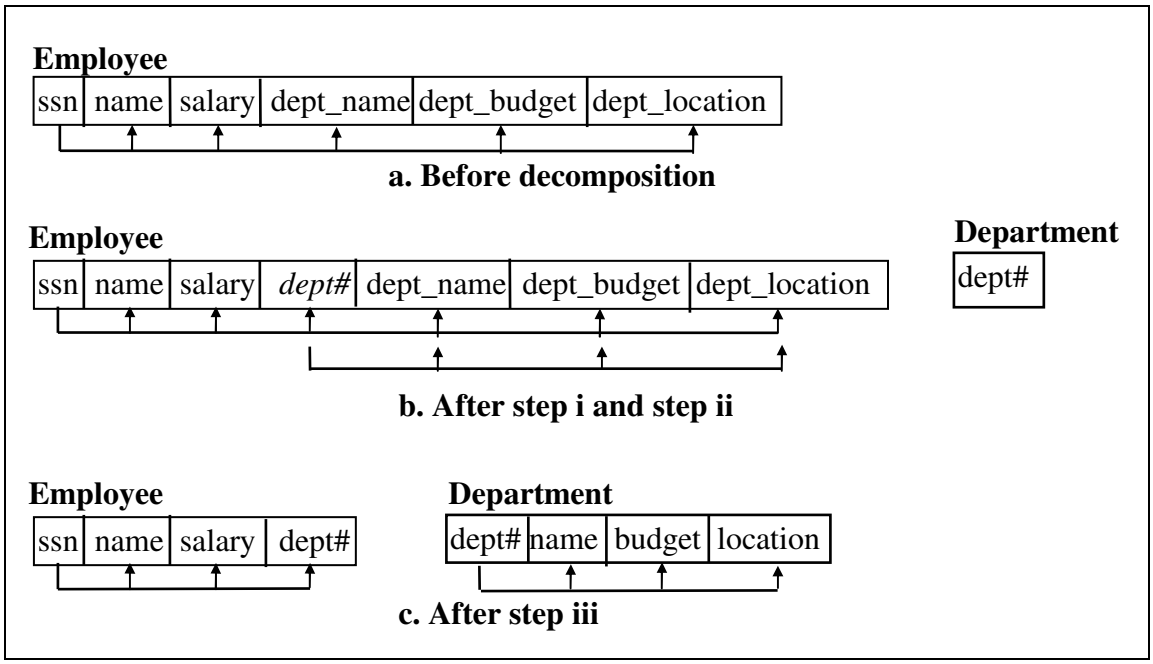


Fig. 2. Decomposition of a relation in the normalization process

Fig. 2 shows an example of decomposing a relation Employee into a modified version of Employee and a new relation Department. A designer first detects update anomalies. For instance, if a department is allocated with a new budget, all tuples of the employees working for the department should be updated. Intrigued by the update anomalies, the designer concludes that there should be some hidden functional dependencies that makes the relation violate the BCNF constraints. Shortly he figures out that the three attributes dept_name, dept_budget, and dept_location, are implicitly dependent on the key attribute of another relation, and name the relation 'Department'. A new relation is created with a primary key dept# and linked to the relation Employee via the newly introduced foreign key dept#. The foreign key then renders the relation Employee not in BCNF. Since dept_name, dept_budget, and dept_location are functionally dependent on dept# and should belong to Department, they are moved from Employee to Department and renamed to name, budget, and location, respectively.

3. Normalization of Object Classes

Having reviewed the normalization of a relation, we are ready to investigate into the normalization of an object class. Our initial concerns are the characteristics of an object class that makes its normalization distinct from that of a relation and the update anomalies we can observe for an object class. After addressing these two issues, we will formalize the object class normalization by introducing the concepts of an object functional dependency and an object

normal form. These concepts will be illustrated with an example.

```
class Employee {
    string name
    set<string> specialties
    Department work_for
    set<Child> dependents
}
```

Fig. 3. An object class Employee

3.1 Characteristics of object classes

The features of an object class can be contrasted with those of a relation as follows.

- Object attributes can be not only simple but also complex. The value of a complex attribute is a reference to the instance of another class. For example in Fig. 3, the name and specialties are simple attributes whereas work_for and dependents are complex attributes.
- Object attributes can be not only single-valued but also multi-valued. Usually, collection types (e.g., set, bag, array, list) are used to denote being multi-valued as well as other semantics such as an ordering. For example in Fig. 3, the attributes name and work_for are single-valued while specialties and dependents are multi-valued.
- Objects are uniquely identified by object identifiers that are assigned by the system. There is no notion of a key attribute at all.

Due to these characteristics, there is no equivalent of relational normal forms for an object class. Both complex attributes and multi-valued attributes make an object class non-1NF. Even if all attributes are simple and single-valued, the lack of a key attribute makes it non-2NF.

3.2 Update anomalies in object classes

Fig. 4 shows an example object class for demonstrating update anomaly problems. Interestingly, unlike the case of a relation, there is no insertion anomaly unless there exists a constraint prohibiting a null on the attribute ssn. For instance, we can insert data about a department even if there is no employee working for the department -- by creating an Employee object and insert only the attributes that are pertinent to the department (dept_name, dept_budget, and dept_locations). However, we can observe anomalies for a deletion and modification to the same extent as we can for a relation.

- Deletion anomalies: For a department with more than one employee, there is no way of removing the department information without deleting all its Employee objects. For a department with only one employee, we inadvertently lose the department data if we delete the Employee object.
- Modification anomalies: In order to change the data about a department, we have to change all the objects of the employees working for the department.

```
class Employee {
    string ssn
    string name
    integer salary
    string dept_name
    integer dept_budget
    set<string> dept_locations
}
```

Fig. 4. An object class with update anomalies

3.3 Object normal form

The existence of an update anomaly problem is sufficient to justify the need for a normalization. Let us first formalize the normalization process of an object class in

analogy to that of a relation. Underlying the relational normalization process is the concept of dependency. Likewise, we need one for an object class.

Definition[*Object functional dependency*]
Given two attributes X and Y that belong to the same object class C, Y is said to be object functionally dependent on X or, equivalently, X is said to determine the value of Y object functionally if and only if the value of Y is determined uniquely for each value of X. This object functional dependency is denoted as X $\bullet\!\!\rightarrow$ Y where Y is a simple or complex attribute that may be of a collection type.

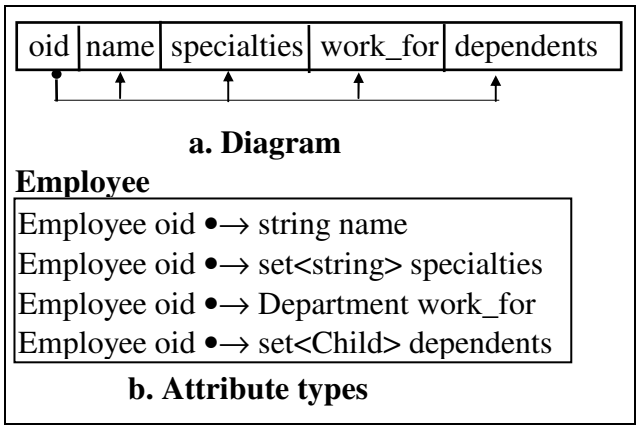
Note that the object functional dependency obviates the need for a multi-valued dependency by virtue of collection types. What would be expressed in pair as X $\rightarrow\!\!\rightarrow$ Y | Z for a relation is specified separately as X $\bullet\!\!\rightarrow$ Y and X $\bullet\!\!\rightarrow$ Z, where Y and Z are attributes of collection types.

Now, we are ready to define the object normal form based on the notion of the object functional dependency.

Definition[*Object normal form*]
An object class C is said to be in the object normal form if and only if all determinants of object functional dependencies are the object identifier (oid) of the class C.

Note that the object identifier is an attribute generated by the system and is invisible to users. Nonetheless, an OODB designer may well assume the existence of an object identifier for a normalization purpose.

The Employee class shown in Fig. 3 is in the object normal form. Its object functional dependencies are shown in Fig. 5. Each attribute is object functionally dependent on the Employee oid. In other words, the Employee oid determines the name, specialties, work_for, and dependents uniquely. The object reference work_for is materialized as the oid of a Department object. An object identifier determines a collection attribute in its entireties, not in individual members. Note that an object normal form looks similar to the BCNF of a relation, except the different notion of a functional dependency and the usage of an object identifier in place of a relational key.



a. Diagram

Employee

Employee oid $\bullet\!\!\rightarrow$ string name
Employee oid $\bullet\!\!\rightarrow$ set<string> specialties
Employee oid $\bullet\!\!\rightarrow$ Department work_for
Employee oid $\bullet\!\!\rightarrow$ set<Child> dependents

b. Attribute types

Fig. 5. Object functional dependencies in the object normal form class of Fig. 3

3.4 Normalization to an object normal form

With the formal notion of an object normal form in mind, an OODB designer can perform a normalization in the same manner as he does for a relation:

i) Create a referenced class if one does not exist.
ii) Introduce an object reference if one does not exist.
iii) Move decomposed attributes to the referenced class. Rename the attributes if necessary.

illustrates the steps of normalizing the non-object normal form class Employee of Fig. 4 into two object normal form classes Employee and Department in b. The decomposition process is similar to that of

a relation. A designer, after detecting update anomalies, decides to create a new class Department and introduce a complex attribute dept in Employee as a reference to a Department object. It is then exposed that the modified
Employee class is not in the object normal form because of the additional object functional dependencies that are shown in Fig. 7. The three attributes, dept_name, dept_budget, and dept_locations, are hence moved to the class Department and renamed to name, budget, and locations, respectively.

4. Summary

In this paper, we addressed the normalization of an object class in designing an OODB schema. Like in a relational database, update anomalies were observed in an unnormalized object class. These update anomalies make it necessary to devise a normalization method for an object class. To formalize the normalization process, we first invented the concept of an object functional dependency. This dependency can be regarded as an integrated object version of both functional dependency and multi-valued dependency used in the normalization of relations. Secondly, we defined the object normal form and its constraints based on the notion of the object functional dependency. Lastly, we showed an example of the decomposition process for normalizing an object class into object normal form classes.

```
class Employee {                    class Department {
    string ssn                      }
    string name
    integer salary
    Department dept
    string dept_name
    integer dept_budget
    set<string> dept_locations
}
```

a. After step i and step ii

```
class Employee {                    class Department {
    string ssn                          string name
    string name                         integer budget
    integer salary                      set<string> locations
    Department dept                 }
}
```

b. After step iii

Fig. 6. Decomposition of an object class in the normalization process

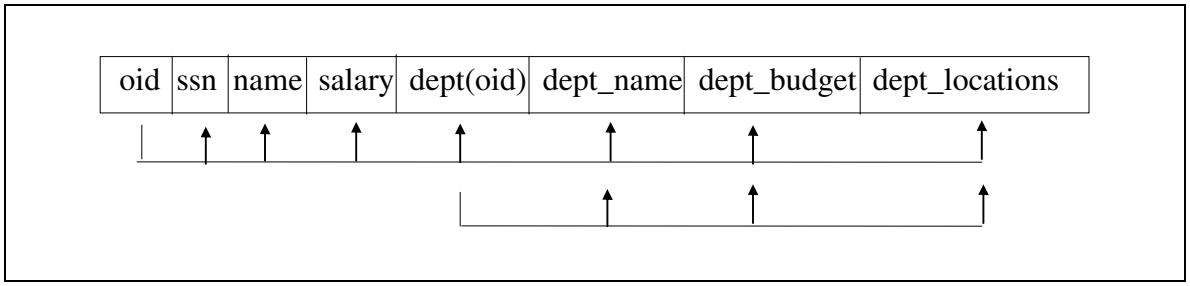


Fig. 7. Object functional dependencies of the Employee class in a

References

[1] P. J. Pratt and J. J. Adamski, *Database Systems Management and Design (3rd edition),* Boyd & Fraser Publishing Company, Danvers, MA, 1994, pp. 597-598.

[2] Pratt P.J. and Adamski, J.J., *Database Systems Management and Design (3rd edition),* Boyd & Fraser Publishing Company, Danvers, MA, 1994, Chapter 6.

[3] Date, C.J., *An Introduction to Database Systems (5th edition),* Addison-Wesley

Publishing Company, Inc., Reading, MA, 1990, Volume I, Chapter 21.

[4] Elmasri, R. and Navathe, S., *Fundamentals of Database Systems (2nd edition),* Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994, Chapters 12 and 13.