Performance evaluation of the object-relational transformation methodology

Johanna Wenny Rahayu a,*, Elizabeth Chang a, Tharam S. Dillon a, David Taniar b

a Department of Computer Science and Computer Engineering, La Trobe University, Bundoora, Vic. 3083, Australia
b School of Business Systems, Monash University, P.O. Box 63B, Clayton, Vic. 3800, Australia

Received 15 February 1999; received in revised form 26 July 2000; accepted 14 March 2001

Abstract

The emergence of the object-oriented (OO) methodology has shown its capabilities in modelling the real world better than the earlier relational methodology. However, object-oriented databases (OODBs) are still considered immature in comparison with relational databases (RDBs) which have existed for many years. RDBs still continue to dominate the implementation of databases constituting more than 90% of all database implementations [28]. It was felt worthwhile to exploit the great modelling power of OO methodology, while still facilitating relational implementations. These reasons have led us to develop an object-relational transformation methodology [20–25] which allows us to use the OO methodology for data modelling and to transform it into a relational logical model for implementation in relational database management systems (RDBMSs). The main purpose of this paper is to present a performance evaluation of the transformation methodology. The evaluation covers I/O cost models of different types of queries. The type of evaluation is basically comparison-based, in which the performance of SQL operations upon a set of tables derived from the relational data model is compared with the tables derived from the OO data model using the transformation methodology. The results of the evaluation show that the performance of the RDB implementation transferred from an OO conceptual model using our object-relational transformation methodology is better than the relational implementation using a conventional relational modelling. Moreover, in many cases, the relational modelling is not applicable since it cannot capture the design semantics particularly relating to collection types. Our object-relational methodology solves this problem. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Object-oriented conceptual model; Object-relational transformation; Relational databases; Performance evaluation; Quantitative analysis; Qualitative analysis

1. Introduction

Object-oriented (OO) concepts offer many features such as extendability, increased modelling power, information hiding and reusability. These are the reasons for the emergence of a new

*Corresponding author.

E-mail addresses: wenny@cs.latrobe.edu.au (J.W. Rahayu), chang@cs.latrobe.edu.au (E. Chang), tharam@cs.latrobe.edu.au (T.S. Dillon), david.taniar@infotech.monash.edu.au (D. Taniar).

0169-023X/01 - see front matter © 2001 Elsevier Science B.V. All rights reserved.

PH: S 0 1 6 9 - 0 2 3 X ( 0 1 ) 0 0 0 2 6 - X
generation of database systems: namely, object-oriented database management systems (OODBMs).

However, these new database systems still need to be refined before they can be integrated into the evolutionary history of database technology. The main reason for the emergence of a new approach in the development of database systems, namely, the object-relational approach, is that relational data modelling has limited capabilities in modelling real-world applications, and that OODBMs are still not as widely used as relational databases (RDBs) that rest on a firm formal foundation. Despite the differences between the OO and the relational paradigm, in reality most object-based development systems are still using the relational database management system (RDBMS) engine as its persistence mechanism [1].

In general, we identify five different categories of object-relational approaches. The first category is the object-relational transformation methodology where OO modelling is used to capture the nature of the problem domain, and then the design is transformed into RDB schemas (tables) for implementation. The object-relational approach discussed in this paper falls into this category. The transformation methodology in our approach does not assume an entirely new methodology; rather it is a methodology that can be used to bridge the existing OO conceptual modelling and the relational logical design for implementation of an object-relational system that utilises an RDBMS as its database engine.

The second category is called object-wrappers (or object-layer) on relational systems, where a layer is built on top of a conventional RDB engine that simulates OO features. One possible aim of this layer is to transform object queries (OQL) submitted by users into relational queries (e.g., open ODB by Hewlett–Packard). Another variation of this category is object-based queries [2,29] where the object layer maps relations into object-views and each object-view can be a complex combination of join and projection operations on the base relations.

The third category is the extended relational systems. In this category, relational systems are extended in order to support OO features. The extensions include: the support of object identifiers (OID), inheritance structures, complex types representations and user-defined operations.

The fourth category is an approach called OO system and RDBMS co-existence. As opposed to a hybrid system in which both OO and relational systems are combined into a single system by adding relational features to an OO system or adding OO features to a relational system, the co-existence approach provides an interface that allows OO systems to access and manipulate a relational system by encapsulating RDB entities such as tables and queries into objects. For example, the recent Borland Database Engine API for Borland C++ Builder allows an OO programming language C++ to access standard data sources in Paradox, dBase or Interbase format.

The fifth category is the OODBMS and RDBMS interoperation. This approach is frequently used in a multidatabase system. A multidatabase system is a database system that controls multiple translators (or gateways) – one translator for each remote database [14]. In this type of environment, it is possible for one application program to work with data retrieved from both an ODBMS and one or more RDBMSs.

It is important to note that whilst the first category serves as the main foundation of the transformation strategies in this paper, the techniques that we have developed can also be utilised by the other categories as a method for efficient and straightforward communication between OO and relational systems.
Our previous work reported in [20–25] presents an object-relational transformation methodology that includes the mapping of inheritance, association and aggregation to relational tables for implementation in any commercial RDBMSs. It is the aim of this paper to present a performance evaluation of the object-relational transformation methodology by comparing the relational implementation derived from the object-relational transformation and that derived from the conventional relational modelling. Fig. 1 shows the two methodologies to be compared and evaluated. The evaluation includes quantitative analysis and comparison of operations in the proposed methodology as well as in the existing RDB design methodology.

The rest of this paper is organised as follows. Section 2 describes the adopted object model, which consists of classes/objects, inheritance, aggregation and association. Section 3 explains the performance evaluation framework. Sections 4–6 present the performance evaluation for inheritance, aggregation and association, respectively. Each of these sections describes our object-relational approach, the conventional relational approach, and makes a comparison. Section 7 discusses related work on object-relational transformation. Finally, Section 8 presents the conclusion and outlines future work.

2. The adopted object model
The adopted object model consists of classes/objects, inheritance, association, and aggregation. A class is a description of several objects that have similar characteristics [7]. An object is an entity that contains both the attributes that describe the state of the object, and the actions that are associated with the objects. Each object has an identity, called object identity (OID). An OID is an
invariant property of an object, which distinguishes it logically and physically from all other objects. An OID is therefore unique. Two objects can be equal without being identical. The state of an object is actually a set of values of its attributes. Actions (often known as methods) are specified as operations, which are defined in the class that describe the object. In this paper, only those attributes and relationships between objects will be considered in the object-relational transformation. This covers classes, inheritance, association and aggregation.

It is convenient to use a graphical notation to represent an object model. We will use a notation that is a modified UML notation. Any modifications will be clarified throughout this discussion. Most of these relate to the semantics and definitions of some terms such as composition, aggregation, etc. A class is often drawn as a rectangle having a class name and specific properties (attributes and methods). With far fewer details, a class is often shown as a square with the class name only. Fig. 2 gives an illustration of a graphical notation for classes.

2.1. Inheritance

An Inheritance relationship is generally known as a generalisation/specialisation relationship, in which the definition of a class can be based on other existing classes. Given that a class inherits from another class, the former class is known as a subclass, whereas the latter is the superclass.

Inheritance can be single or multiple inheritance. Single inheritance is where each subclass has only one superclass. On the other hand, multiple inheritance is where a subclass inherits from multiple superclasses.

Inheritance can be grouped into several types according to the semantics of the relationship. Sometimes we wish to state that the objects in the subclasses constitute all the objects in the superclass. We may also wish to state that the subclasses are mutually exclusive or that they form a partition of the superclass. In this context, inheritance can be union inheritance, mutual exclusion inheritance or partition inheritance.

In terms of its notation, we use an empty arrowhead directed arc for union inheritance, and a filled black coloured arrowhead directed arc for mutual exclusion and partition inheritance. Union inheritance is also known as overlap inheritance [9], and mutual exclusion and partition inheritance are known as disjoint inheritance [9]. In our paper, we specifically distinguish these two disjoint inheritances mutual exclusion and partition inheritance. A label is added to the notation

![Diagram of a class](image)

Fig. 2. A class.
to differentiate mutual exclusion from partition inheritance. Without this, the designers may have to infer a specific inheritance type from data distribution.

*Union inheritance* declares that the union of a group of subclasses constitutes the entire membership of the superclass. When we have a union type of inheritance, we know that every object in the superclass is an object of at least one of the subclasses [10]. Fig. 3(a) gives an example of union inheritance. In this example, it ensures that a Person is either a Student or a Staff member. However, the union type does not preclude a member of a subclass from being a member of another subclass. For example, a Person who is a Staff member may also be a Student. Therefore, this union type is often called overlap inheritance.

A *mutual exclusion inheritance* declares that a group of subclasses in an inheritance relationship are pairwise disjoint. An example of the mutual exclusion type of inheritance is shown in Fig. 3(b). This example is called mutual exclusion because there is no Manager who is also a Worker, and

---

**Fig. 3. Inheritance.**
vice versa. However, in this case there may be an Employee who is neither a Manager nor a Worker.

Partition inheritance declares that a group of subclasses partitions a superclass. A partition requires that the partitioning sets be pairwise disjoint and their union constitutes the partitioned set. Therefore, a partition type can be said to be a combination of union and mutual exclusion types. Fig. 3(c) shows an example of a partition type of inheritance. We use the employee example again, but here a new class Casual is added, and it is assumed that each member of the Employee class must belong to one and only one of the classes: Manager, Worker or Casual. Therefore, it becomes a partition inheritance. For example, an Employee cannot be both Manager and Casual.

Fig. 3(d) gives an example of multiple inheritance. A Tutor class can be said to be inheriting from overlapping classes, because basically a Tutor is a Student who is also a Staff.

2.2. Aggregation

Aggregation is a composition (part-of) relationship, in which a composite object ("whole") consists of other component objects ("parts"). This relationship is used extensively in the areas of engineering, manufacturing and graphics design. In these applications, when a composite object is created, one may merely want to know the type osstf the parts involved, without being bothered with the details. At other times, one may need the details of a particular part only [7].

Dittrich [8], Dillon and Tan [7] and Kim [15] identify four types of composition: sharable dependent, sharable independent, non-sharable dependent, and non-sharable independent composition. We will refer to non-sharable and sharable as exclusive composition and non-exclusive composition, and dependent and independent as existence dependent and existence independent composition, respectively. We use composition interchangeably with aggregation and use qualifications to distinguish between the different categories. Additionally, our model also includes two more types of aggregation relationships, i.e., ordered composition and homogeneous/heterogeneous composition.

In UML notations, only two types of aggregation are identified [4]. The first, called simple aggregation, is basically a relationship between a class that represents a larger thing (the whole) which consists of smaller things (the parts). There are no further semantics attached to this simple type of aggregation. This type of aggregation is represented by a blank diamond that is attached to the whole object. The second type, called composition, is a more tightly coupled aggregation. In composition type, the existence of the parts is entirely dependent on the existence of the whole class, and the one particular part may only be a part of one composition at a time. This composition type is represented by a diamond filled with black colour. We will adopt these UML notations for our different aggregation types, with some additional notations to represent the different semantics we have that are not captured using the original UML notations. For example, in an ordered type of aggregation, we add a square-bracket index number to specify the indexing constraints of the part classes. Whenever the aggregation relationship is very tightly coupled (i.e., existence dependent and exclusive), we will adopt the UML notation for composition.

Existence dependent and existence independent compositions are two aggregation types in which the dependencies between the whole object and its part objects are significant. In an existence-dependent type of composition, the existence of the part is fully dependent on the existence of the whole object. Fig. 4(a) shows an example of an existence-dependent composition. In this example, a Building is an encapsulation of several part objects, i.e., Door, Wall and Window. When a
Building is accessed, its part objects can be identified without the necessity to trace every link from the Building object. Existence dependent forces the deletion of a Building to cause the deletion of that particular Building and all of its elements.

In an existence independent type of composition, the existence of the part is independent. For example (see Fig. 4(b)), if for some reasons the Lab is removed, the Computer, Printer and Scanner still exist.

A whole object may be composed of different part objects in a particular order. For example, a Book is composed of Preface, Chapter and Appendix; exactly in this order. In other words, the order of occurrence of the part objects in the composition is significant to the model. In the composition representation, the square brackets, which indicate constraints, show the ordering of the part object types. Fig. 4(c) gives an example of an ordered aggregation.

Creating an exclusive composition means that the whole object is the sole owner of the part objects. The need for exclusiveness arises when modelling physical objects, such as cars or aeroplanes. In order to capture the semantics of such applications, the aggregation should permit an enforcement of the fact. For example, cars do not share engines or bodies [12]. In the example
shown in Fig. 4(d), we need to ensure that every part object is exclusively owned by a particular whole only.

In a non-exclusive composition, a part of one whole object may be shared or referenced by other whole objects, and thus the part is not exclusive. For example, a BinaryFile or a TextFile can be referenced by more than one Directory (see Fig. 4(e)).

The previous examples are categorised into a heterogeneous aggregation, since one whole object may consist of several different types of part objects. In contrast, homogeneous aggregation implies that one whole object consists of part object/s that is of the same type. For example, a DailyProgram consists of several Programs or an Appointment List consists of several Appointments (see Fig. 4(f)). The main advantage of modelling the homogeneous type of composition is that the model is flexible enough for further extension or modifications to include components of another type. In the case of a mixture of homogeneous and heterogeneous components, the homogeneous composition is indicated by the cardinality, namely, 1 to \( m \).

2.3. Association

Association refers to a “connection” between object instances. Association is basically a reference from one object to another that provides access paths among objects in a system. For example, a Student and a Department are connected through an “EnrolsIn” association link. The link can have a specific cardinality, e.g., one-to-one, one-to-many and many-to-many. In addition to this, in object-orientation, collection types have also been introduced and can characterise an association link.

ODMG defines four different collection types [6]. They are sets, lists, arrays and bags. Sets are basically unordered collections that do not allow duplicates. The objects that belong to a set are all unique. To ensure this constraint, the set insertion operation will verify each object passed as its argument. If the object is already a member of the set, the insert operation will not be executed.

Lists are ordered collections that allow duplicates. The order of the elements in a list is based on the insertion order. Thus, the database should be able to keep track of the index of each element in each insert or delete operation executed on the list. Arrays are one-dimensional arrays with variable length in terms of the OO data model. An array will be implicitly extended by assignments to array elements beyond the current end of the array. Deletion of an array element value does not lead to contraction of the array or to a change of index. The main difference between a list and an array is in the method used to store the pointers that assign the next element in the list/array. Because this difference is mainly from the implementation point of view, lists and arrays will have the same transformation procedure. In this paper, lists will be used to refer to both lists and arrays.

A Bag is similar to a set, except that it allows duplicate values to exist. Thus, it is an unordered collection that allows duplicates. Unlike sets, insertion operations of bags do not check whether an incoming object will cause any duplication.

In UML notations, association relationships are represented as a line connecting the participating classes, and may have a name [4]. The cardinalities between the classes (called multiplicity) are placed at the opposite ends of the relationship. Since in UML, cardinality is only represented as a number (e.g., 1, 2, \ldots\) or many (drawn as an asterisk) and no specific representation for collection types, whenever the cardinality of a relationship involves collection types, we use the collection type name as the cardinality of the association. The collection type name is placed in
between curly brackets to avoid confusion with other textual information related to the association. Fig. 5 shows an example.

3. Performance evaluation framework

Our object-relational transformation methodology, as discussed in [20–24], covers the mapping of inheritance, aggregation, and association; the three important components of an object model as described in the previous section. The scope of our object-relational transformation methodology is shown in Fig. 6.

The performance evaluation, the focus of this paper, as shown in Fig. 1, is that for each of the mapping strategies (i.e., inheritance, aggregation and association), we present our approach and the conventional relational approach. We compare the two approaches by examining the query efficiency to data that are stored in relations derived using our object-relational transformation and comparing this to relational modelling.

In order to evaluate the effectiveness of the implementation of each methodology, it is necessary to provide cost models that will be used to perform quantitative analysis. Since our purpose is to compare (not to estimate the total cost), some simplifications have been made to the cost models without affecting the comparison results. The calculation is mainly based on the number of I/O accesses, as I/O access is considered much more expensive than the CPU processing cost (e.g., storage cost, computation cost, etc.).

Each relational table $T$ has a set of records $r = \{r_1, r_2, \ldots, r_n\}$ which are stored in a set of pages $p = \{p_1, p_2, \ldots, p_m\}$. Assume that the relational tables involved in a query operation are scanned separately. The equation of retrieval cost is as follows:

![Object-Relational Transformation Methodology](image-url)
Page cost = \( \text{Round} \left( \frac{R}{\text{Trunc} \left( \frac{\text{page size}}{S} \right)} \right) \approx \left( \frac{R \cdot S}{\text{page size}} \right) \), \hspace{1cm} (1)

where \( R \) is the number of records and \( S \) is the record size.

Performance evaluation is assessed only according to the number of pages accessed (I/O accesses). Eq. (1) suggests that two important parameters for the performance evaluation are record size and number of records.

4. Performance evaluation of inheritance

In this section, we would like to present performance evaluation of the object-relational transformation methodology, particularly relating to mapping inheritance.

4.1. Object-relational approach for mapping inheritance

The object-relational transformation methodology described here is mainly based on our previous work reported in [20–24]. In order to simplify the description of the methodology, only a summary of the rules will be presented.

Each class in the OO model can be directly transformed into a relational table. The concept of object identity is preserved by including an extra unique attribute in the relational table, serving the same function as a surrogate or primary key.

Since there are different types of inheritance, each type will be treated differently in the transformation process.

Mapping union inheritance. The superclass and each of the subclass/es are mapped to relational tables. A share ID is used to preserve the identity of the objects across tables. Using the Person–Staff–Student example shown in Fig. 3(a), the result of the transformation of union inheritance is:

- Person(ID, Name, Address)
- Student(ID, Course, Year)
- Staff(ID, Department, Room)

Mapping mutual exclusive (disjoint) inheritance. To transform this type of inheritance, a new attribute, namely, “Subtype” is added in the superclass table. This Subtype ensures every member of the subclass tables is disjoint. Using the Employee–Manager–Worker example shown in Fig. 3(b), the result of the transformation is:

- Employee(ID, Name, Address, Subtype)
- Manager(ID, AnnualSalary)
- Worker(ID, WeeklyWage)

Mapping partition (union disjoint) inheritance. This type can be seen as a combination of both union and mutual exclusion. In the transformation to tables, a new attribute Subtype is also added as in mapping mutual exclusion. The difference is that with mutual exclusion null values are allowed in the Subtype field in this partition type. The not null Subtype field will ensure that every member of the superclass table is also a member of a subclass table. Using the Employee–
Manager–Worker–Casual example shown in Fig. 3(c), the result of the transformation is as follows:

\[
\begin{align*}
\text{Employee}(\text{ID}, \text{Name}, \text{Address}, \textbf{Subtype}) & \quad \textbf{Subtype NOT NULL} \\
\text{Manager}(\text{ID}, \text{AnnualSalary}) \\
\text{Worker}(\text{ID}, \text{WeeklyWage}) \\
\text{Casual}(\text{ID}, \text{HourlyRate}) \\
\end{align*}
\]

**Mapping multiple inheritance.** To transform multiple inheritance with overlapping classes into relational tables, a table is created for every super/subclass. A single ID is used to preserve the relationship between the super/subclass tables. Using the Person–Staff–Student–Tutor example in Fig. 3(d), the transformation result is as follows:

\[
\begin{align*}
\text{Person}(\text{ID}, \text{Name}, \text{Address}) \\
\text{Student}(\text{ID}, \text{Course}, \text{Year}) \\
\text{Staff}(\text{ID}, \text{Department}, \text{Room}) \\
\text{Tutor}(\text{ID}, \text{NoHours}, \text{Rate}) \\
\end{align*}
\]

### 4.2. Relational modelling approach

In the original relational modelling, there is no notion of either inheritance or abstract entities. Since this pioneer work, there have been a number of attempts to include subtyping in the relational modelling. Using these conventional approaches, we summarise three ways of implementing inheritance. The first way is “one table each”, the second way is “all in one table” and the last way is “subtyping”. From now on, we will refer to them as Type 1, Type 2 and Type 3, respectively. To distinguish our method from Type 1, Type 2 and Type 3, we will call our method object-relational (OR).

#### 4.2.1. Type 1: one table each

Let the inheritance schema shown in Fig. 7 be an example. The Type 1 approach creates each class node as an entity, complete with its attributes. The corresponding ER schema is shown in Fig. 8. Furthermore, if class Customer is not abstract, we also create Customer as an entity (see Fig. 9).

![Customer inheritance schema](image-url)
![ER Diagram](image1.png)

Fig. 8. ER diagram where customer entity is abstract.

![Customer Diagram](image2.png)

Fig. 9. Non-abstract customer entity.

The obvious consequences of this modelling include repeated attributes, and repeated records (for overlap inheritance only). More importantly, this modelling will not be efficient if there is an association to/from the superclass, such as an association between Customer and another entity called Supplier (see Fig. 10(a)). The transformed ER diagram will look like that shown in

![Object Conceptual Model](image3.png)

Fig. 10. (a) Object conceptual model; (b) ER model.
Fig. 10(b). If the association relationship is one-to-many, we need to store the Primary Key (PK) of the superclass as a Foreign Key (FK) in each of the other tables (i.e., Customer, Commercial, Academic, Private). If it is the opposite (i.e., many-to-one relationship), each of the PKs of Customer, Commercial, Academic, and Private become FKS in the Supplier. If it is a many-to-many, we need to have four additional tables, which maintain the association relationship. This is certainly not efficient because multiple attributes must exist to represent a single association relationship. Hence, this method is usually not used by designers. We eliminate this type from further analysis.

4.2.2. Type 2: all in one table

All subclass nodes are collapsed within their superclass node. This is a naive approach, but applicable in some cases where subclass nodes have few additional attributes/methods. In the entity, a “type” attribute is usually included to indicate the type of the object/record. Fig. 11 shows a one-entity model.

The implementation is simple and easy with an effect that null values often occur in the specialised attributes (e.g., ACN, department, DOB).

4.2.3. Type 3: subtyping

This type is similar to Type 1, but includes a link between subclasses to their superclass by having the same ID key. In other words, each class becomes a table, and each table, apart from having attributes declared in the class, is added with an attribute called ID to store the OID of each object (record). The ID attribute subsequently becomes the PK of each table. This subtyping model is a manifestation of a “vertical division”, where an object is divided vertically into a number of classes. The table’s structure becomes:

Customer(ID, Name, Address)
Commercial(ID, ACN)
Academic(ID, Department)
Private(ID, DOB)

This type is similar to our transformation strategy (union inheritance), but we have provided more comprehensive rules, which include mutual exclusion inheritance, partition inheritance, etc.

4.3. Cost models

In the cost models, we particularly consider record size (S) and number of records (R).

4.3.1. Record size (S)

The record sizes for Types 2, 3 and object-relational can be calculated as follows.

![Diagram of one entity model](image)
Type 2 ($S_{T2}$). The record size is the sum of superclass attributes and subclasses attributes. Additionally there is an ID attribute and a type attribute:

$$S_{T2} = S_{oid} + S_{type} + S_1 + S_2 + \cdots + S_m = S_{oid} + S_{type} + \sum_{i=1}^{m} S_i,$$

(2)

where $S_2$ to $S_m$ are the size of the specialised attributes, and the inheritance schema involves $m$ number of classes, with $i = 1$ being the superclass.

For example, the following are sample attributes of the four classes: Customer, Commercial, Academic and Private:

- $Customer(S_1) = \text{name, address}$
- $Commercial(S_2) = \text{ACN}$
- $Academic(S_3) = \text{department}$
- $Private(S_4) = \text{DOB}$.

Since Type 2 ($S_{T2}$) is an “all in one table”, there will be one table to represent the above inheritance hierarchy. The table will have all of the above attributes, plus two additional attributes, namely, ID and Type. The ID attribute stores the OID of each object (record), whereas the Type attribute stores the type of each object (record). The attributes for $S_{T2}$ are as follows:

- **all in one table** ($S_{T2}$) = ID, name, address, ACN, department, DOB, type,

($S_{T2}$) = $S_{oid} + S_1 + S_2 + S_3 + S_4 + S_{type}$.

In general, Eq. (2) is used to calculate $S_{T2}$.

Type 3 ($S_{T3}$). The record size is individual to each table/class. Hence,

$$S_{sup} = S_1 + S_{oid},$$

$$S_{sub1} = S_2 + S_{oid},$$

$$S_{sub2} = S_3 + S_{oid},$$

$$\vdots$$

$$S_{sub(m-1)} = S_m + S_{oid},$$

(3)

where $S_{sup}$ is the size of a superclass object and $S_{sub}$ is the size of a subclass object.

Using the above Customer–Commercial–Academic–Private inheritance schema as an example, each class becomes a table. Each table will have the attributes declared locally in each class, plus an additional attribute called ID to store the OID of each object. Hence,

- $Customer(S_{sup}) = \text{ID, name, address}$
- $Commercial(S_{sub1}) = \text{ID, ACN}$
- $Academic(S_{sub2}) = \text{ID, department}$
- $Private(S_{sub3}) = \text{ID, DOB}$.

The size of the ID attribute is $S_{oid}$, whereas the size of attributes declared in class $i$ is $S_i$. Assume class 1 is the superclass, whereas other classes are the subclasses. For example, $S_1$ is the size of
name and address (class Customer), \(S_2\) is the size of ACN (class Commercial), \(S_3\) is the size of department (class Academic) and \(S_4\) is the size of attribute DOB (class Private). In general, Eq. (3) is used to calculate the record sizes for Type 3 (\(S_{T3}\)).

Object-relational (\(S_{oi}\)). Our approach is similar to \(S_{T3}\). However, if it is a mutual exclusion or partition inheritance, \(S_{sup}\) becomes

\[
S_{sup} = S_1 + S_{oid} + S_{type}.
\]

The type attribute for mutual exclusion inheritance is used to determine the type of the object, whilst in partition inheritance it is used to ensure that the type is not null. Using the above example, class Customer becomes

\[
Customer(S_{sup}) = \text{OID, name, address, Subtype}
\]

As the size of the type attribute is \(S_{type}\), Eq. (4) can be used to calculate the size of the superclass table (\(S_{sup}\)).

4.3.2 Number of records (\(R\))

The number of records for Type 2, 3 and object-relational can be calculated as follows.

Type 2 (\(R_{T2}\)). Number of records is the sum of all objects of the superclass and its subclasses:

\[
R_{T2} = \sum_{i=1}^{m} R_i,
\]

where \(R_1\) is the number of superclass records and \(R_k\) (\(2 \leq k \leq m\)) is the number of subclass records. To clarify the equation, we provide some sample data for the table, which is shown in Fig. 12.

Since class Customer is an abstract class, \(R_1 = 0\) (no instances). Number of commercial objects is \(R_2 = 2\), number of academic objects is \(R_3 = 4\) and number of private object is \(R_4 = 1\). Therefore, \(R_{T2} = 0 + 2 + 4 + 1 = 7\) records. In general, Eq. (5) is used to calculate number of records of the table \(R_{T2}\).

We need to emphasise that the value of the Subtype attribute in the table in Fig. 12 could have been compressed to, for example, one character, instead of the full string. Or it can be incorporated with the OID. Whichever method is adopted, in the cost model, the Subtype attribute will have a certain length and will be occupied by some values, which indicate the Subtype (or subclass) of the record.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Address</th>
<th>ACN</th>
<th>Department</th>
<th>DOB</th>
<th>CustType</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Myer Pty Ltd.</td>
<td>Melbourne</td>
<td>123-423</td>
<td></td>
<td>Commercial</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Coles Pty Ltd.</td>
<td>Sydney</td>
<td>443-765</td>
<td></td>
<td>Commercial</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>LaTrobe Univ.</td>
<td>Bundoora</td>
<td></td>
<td>Comp. Sc.</td>
<td>Academic</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Monash Univ.</td>
<td>Gippsland</td>
<td></td>
<td>Info. Tech</td>
<td>Academic</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>RMIT Univ.</td>
<td>Melbourne</td>
<td></td>
<td>Comp. Sc.</td>
<td>Academic</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Victoria Univ.</td>
<td>Footscray</td>
<td></td>
<td>Informatics</td>
<td>Academic</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Rahayu</td>
<td>Kew</td>
<td></td>
<td></td>
<td>05/05/63</td>
<td>Private</td>
</tr>
</tbody>
</table>

Fig. 12. Sample data for Type 2 model.
Type 3 ($R_{T3}$). The number of records for this model is individual to each class, because each class in the hierarchy is a table. Hence, $R_2$ to $R_m$ are the number of subclass records. However, $R_{sup}$, the number of superclass records is the same as $R_{T2}$, which is the sum of $R_1$ to $R_m$ and $R_1$ is number of non-specialised superclass objects:

$$R_{sup} = R_{T2} = \sum_{i=1}^{m} R_i.$$  \hspace{1cm} (6)

To clarify Type 3, we use the tables in Fig. 13.

Since all subclass objects are superclass objects as well, all objects from class Commercial, Academic and Private are Customer objects. The number of Customer table in the above-mentioned example is $R_1 = 7$. The number of records in each subclass table is the number of objects in each subclass: 2 commercial objects ($R_2 = 2$), 4 academic objects ($R_3 = 4$) and 1 private object ($R_4 = 1$).

Object-relational ($R_{or}$). $R_{or}$ is the same as $R_{T3}$. The number of records of the superclass table in $R_{or}$ is the same as that of $R_{T3}$, though the structure of the table is different, since in $R_{or}$, an additional attribute called Subtype is added to the superclass table. Nevertheless, the number of records is the same. See the Customer table in Fig. 14.

![Customer Table](image1)

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Myer Pty Ltd.</td>
<td>Melbourne</td>
</tr>
<tr>
<td>2</td>
<td>Coles Pty Ltd.</td>
<td>Sydney</td>
</tr>
<tr>
<td>3</td>
<td>LaTrobe Univ.</td>
<td>Bundoora</td>
</tr>
<tr>
<td>4</td>
<td>Monash Univ.</td>
<td>Gippsland</td>
</tr>
<tr>
<td>5</td>
<td>RMIT Univ.</td>
<td>Melbourne</td>
</tr>
<tr>
<td>6</td>
<td>Victoria Univ.</td>
<td>Footscray</td>
</tr>
<tr>
<td>7</td>
<td>Rahayu</td>
<td>Kew</td>
</tr>
</tbody>
</table>

![Commercial Table](image2)

<table>
<thead>
<tr>
<th>ID</th>
<th>ACN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>123-423</td>
</tr>
<tr>
<td>2</td>
<td>443-765</td>
</tr>
</tbody>
</table>

![Academic Table](image3)

<table>
<thead>
<tr>
<th>ID</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Comp. Sc.</td>
</tr>
<tr>
<td>4</td>
<td>Info. Tech</td>
</tr>
<tr>
<td>5</td>
<td>Comp. Sc.</td>
</tr>
<tr>
<td>6</td>
<td>Informatics</td>
</tr>
</tbody>
</table>

![Private Table](image4)

<table>
<thead>
<tr>
<th>ID</th>
<th>DOB</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>05/05/63</td>
</tr>
</tbody>
</table>

![Customer Table](image5)

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Address</th>
<th>Subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Myer Pty Ltd.</td>
<td>Melbourne</td>
<td>Commercial</td>
</tr>
<tr>
<td>2</td>
<td>Coles Pty Ltd.</td>
<td>Sydney</td>
<td>Commercial</td>
</tr>
<tr>
<td>3</td>
<td>LaTrobe Univ.</td>
<td>Bundoora</td>
<td>Academic</td>
</tr>
<tr>
<td>4</td>
<td>Monash Univ.</td>
<td>Gippsland</td>
<td>Academic</td>
</tr>
<tr>
<td>5</td>
<td>RMIT Univ.</td>
<td>Melbourne</td>
<td>Academic</td>
</tr>
<tr>
<td>6</td>
<td>Victoria Univ.</td>
<td>Footscray</td>
<td>Academic</td>
</tr>
<tr>
<td>7</td>
<td>Rahayu</td>
<td>Kew</td>
<td>Private</td>
</tr>
</tbody>
</table>

Fig. 13. Sample data for Type 3 model.

Fig. 14. Sample data for object-relational model.
As in Type 2, the value of the Subtype attribute in this example is a full string. *It could have been shortened into one byte, instead, or incorporated into the OID field.* If the latter is applied, we have to make sure that all OIDs of the records in the other tables must be of the same format, since OID should be universal across the tables in a database.

4.4. Analysis

Queries featuring inheritance can be categorised into two queries: *superclass* and *subclass.* *Superclass queries* are queries accessing general attributes declared in the superclass, whereas *subclass queries* are queries accessing specific attributes declared in the subclass.

4.4.1. Superclass queries

Let us consider a partition inheritance. Suppose that each customer should belong to one and only one of the subclasses *Commercial, Academic* and *Private.*

**Query 1.** An example of a superclass query is “to retrieve the names and addresses of all academic customers”. This query is categorised as a superclass query because the query accesses general attributes declared in the superclass, namely, *name* and *address.*

The SQL for this query using Type 2 (SQL$_{T2}$) and the proposed object-relational transformation (SQL$_{or}$) is identical which is as follows:

SQL$_{T2}$ = SQL$_{or}$:

Select CUSTOMER.Name, CUSTOMER.Address
From CUSTOMER
Where CUSTOMER.CustType = 'Academic';

However, using Type 3 (SQL$_{T3}$), because there is no partition concept in existing subtyping, we must join superclass table Customer with subclass table Academic. The reason for such a join is that by using the Type 3 model there is no Subtype attribute in the superclass table. As a result, to obtain the above query result, we need to join the tables:

SQL$_{T3}$:

Select CUSTOMER.Name, CUSTOMER.Address
From CUSTOMER, ACADEMIC
Where CUSTOMER.ID = ACADEMIC.ID;

**Quantitative analysis.** It is essential to model the behaviour of superclass queries by providing general cost models, so that the evaluation is not based only on one particular query as above. In addition, later in the section we provide a calculation based on query examples.

For Type 2 and object-relational transformation, the processing of superclass queries is localised to one table only; that is the one table for Type 2 and the superclass table for the object-relational method. Therefore, the processing cost models for both methods (C$_{T2}$ and C$_{or}$, respectively) are as follows:

\[
C_{T2} = \frac{R_{T2} \times S_{T2}}{\text{page size}} = \frac{\sum_{i=1}^{m} R_i \times (S_{\text{oid}} + S_{\text{type}} + \sum_{i=1}^{m} S_i)}{\text{page size}},
\]

\[
C_{or} = \frac{R_{or} \times S_{or}}{\text{page size}} = \frac{R_{\text{sup}} \times S_{\text{sup}}}{\text{page size}} = \frac{\sum_{i=1}^{m} R_i \times (S_{\text{oid}} + S_{i} + S_{\text{type}})}{\text{page size}}.
\]
Since $R_{sup} = R_{T2}$ (see Eq. (6)), the difference between $C_{T2}$ and $C_{or}$ (see Eqs. (7) and (8)) lies in the size of the table.

However, for $C_{T3}$, a join operation must be performed. A join operation is known to be one of the most expensive operations in RDB processing. Compared with a single table access, a join operation is much more expensive. Even without substantial proof, this fact is universally known. Nevertheless, in the following we are trying to highlight the difference between $C_{T2}$ and $C_{or}$ (single table access) and $C_{T3}$ (join).

In the join operation ($C_{T3}$), we assume that a hash join algorithm is used [19], because the hash join algorithm is the most efficient due to its linear complexity (i.e., $O(N)$, where $N$ is the problem size normally measured by number of records) [16]. Using a hash join algorithm, first load and hash records from the first table. Once the hash table is constructed from the first table, the second table is loaded and hashed/probed using the same hashing function. Any matching belongs to the query results.

Depending on the query, there may be a need to join more than two tables. Such a query may to retrieve names of academics and commercial customers, for example. Our object-relational transformation method and Type 2 method still access the one table, whereas Type 3 joins the superclass table with the concerned subclass tables.

**Lemma 1.** For superclass queries, the proposed object-relational transformation methodology is more efficient than the other two.

**Proof.** It has to prove that $C_{or} < C_{T2}$ and $C_{or} < C_{T3}$.

First, let us prove that

$$C_{or} < C_{T2},$$

$$\frac{R_{sup} \times S_{sup}}{\text{page size}} < \frac{R_{T2} \times S_{T2}}{\text{page size}}.$$ 

According to Eq. (6), $R_{sup}$ on the left-hand side of the above equation is the same as $R_{T2}$ on the right-hand side, and hence the above equation becomes

$$S_{sup} < S_{T2},$$

$$(S_{oid} + S_1 + S_{type}) < \left(S_{oid} + S_{type} + S_1 + \sum_{i=2}^{m} S_i \right),$$

$$0 < \sum_{i=2}^{m} S_i.$$ 

Eq. (9) is always true because the right-hand side terms are positive. Hence, $C_{or} < C_{T2}$ is true. In fact, the difference between $C_{or}$ and $C_{T2}$ grows proportionally with the increase of subclass record size ($S_i$).

Now, let us prove that $C_{or} < C_{T3}$.

For simplicity of the join operation, we assume that the tables to be joined are small enough so that the hash table will fit in perfectly in the main memory. This assumption is to make the join operation more efficient, in order to load each table once only. In real life, this assumption may
not stand, particularly if the tables to be joined are very large, causing the hash table to be larger than the available memory, in which multiple loading of the tables has to be done.

We will show that even with this simplest join operation, our object-relational transformation methodology is still more efficient. With a more complex join as indicated above, our proposed methodology will be far superior.

Assume that only two tables are to be joined following the query example 1 above. Now, it is to prove that

\[
C_{or} < C_{T3},
\]

\[
\frac{R_{sup} \times S_{sup}}{\text{page size}} < \frac{R_{sup} \times S_{sup}}{\text{page size}} + \frac{R_2 \times S_{sub1}}{\text{page size}}.
\]

As mentioned in Eq. (4) earlier, \( S_{sup} \) in \( C_{or} \) is actually \( S_1 + S_{oid} + S_{type} \) and \( S_{sup} \) in \( C_{T3} \) is without \( S_{type} \), hence the above becomes

\[
R_{sup} \times (S_{oid} + S_1 + S_{type}) < R_{sup} \times (S_{oid} + S_1) + R_2 \times (S_{oid} + S_2).
\]

Since both \( S_{type} \) and \( S_{oid} \) are normally small and negligible, compared with the original record sizes indicated by \( S_1 \) and \( S_2 \), the above equation becomes

\[
(R_{sup} \times S_1) < (R_{sup} \times S_1) + (R_2 \times S_2).
\]

Eq. (10) is true as the left-hand side terms exist as a subset in the right-hand side terms. The difference between \( C_{or} \) and \( C_{T3} \) can be even larger when more than two tables are involved in the query. In this case, \( C_{T3} \) involves all tables. \( \square \)

**Example 1.** To elaborate and clarify, consider the following example used to calculate the costs of query 1 above.

Suppose the record size for Customer, Commercial, Academic and Private \( S_1 = 0.4 \) Kb, \( S_2 = 0.15 \) Kb, \( S_3 = 0.12 \) Kb, \( S_4 = 0.14 \) Kb and number of records are \( R_1 = 0 \) (this is partition inheritance), \( R_2 = 800 \), \( R_3 = 150 \) and \( R_4 = 600 \). Assume that \( S_{oid} \) and \( S_{type} \) are 0.01 Kb each. Assume each table page is 2 Kb [3], and a record may not span more than a single page.

The costs are \( C_{T2} = (0 + 800 + 150 + 600) \times (0.4 + 0.15 + 0.12 + 0.14 + 0.01 + 0.01)/2 = 1550 \times (0.81 + (0.01 \times 2))/2 = 644 \) I/O accesses, \( C_{T3 join} = ((1550 \times (0.4 + 0.01)) + (800 \times (0.15 + 0.01)))/2 = (636 + 128)/2 = 382 \) I/O accesses and \( C_{or} = 1550 \times (0.4 + 0.01 + 0.01)/2 = 326 \) I/O accesses.

In this case, \( C_{or} \) is twice as low as \( C_{T2} \) and approximately 17% better than \( C_{T3} \). Notice in this case that \( C_{T3} \) is better than \( C_{T2} \) because in \( C_{T3} \) only two tables are involved, whereas in \( C_{T2} \) the big table includes everything from all classes. \( C_{T3} \) also assumes that the entire hash table fits into memory, which prevents repeated access to the tables.

**4.4.2. Subclass queries**

Let us consider the same Commercial, Academic and Private Customer schemas presented earlier in Fig. 7. We are going to present an analysis for a subclass query.

**Query 2.** An example of a subclass query is as follows: “List all ACN of all of the commercial customers”. This is a subclass query, as the query evaluates specialised attributes of a subclass.
The SQL statements for Type 2, Type 3 and object-relational (SQL\textsubscript{T2}, SQL\textsubscript{T3} and SQL\textsubscript{or}, respectively) are as follows:

**SQL\textsubscript{T2}**:  
Select ACN  
From CUSTOMER  
Where CustType = ‘Commercial’;

**SQL\textsubscript{T3} = SQL\textsubscript{or}**:  
Select ACN  
From COMMERCIAL;

Notice that all methods access one table only: Type 2 accesses the big table, whereas Type 3 and object-relational transformation access the subclass table only, which is much smaller than the big table of Type 2.

**Quantitative analysis.** Since Type 3 and the object-relational transformation methods access a smaller subclass table, intuitively, these methods outperform Type 2 method.

**Lemma 2.** For subclass queries, object-relational is as good as Type 3, but is always better than Type 2.

**Proof.** It has to be proven that $C_{or} < C_{T2}$ and $C_{or} = C_{T3}$.

Let us first prove that $C_{or} \approx C_{T3}$. $C_{T2}$ uses Eq. (7), whereas $C_{T3}$ and $C_{or}$ are given as follows.

$$C_{or} = C_{T3} = \frac{R_{sub} \times S_{sub}}{\text{page size}}.$$  \hfill (11)

As indicated by the above equation, $C_{or}$ and $C_{T3}$ are equal. The number of records in a subclass of both methods is determined by the specialised records of that particular subclass, and hence both $R_{sub}$ are the same. The record size of the subclass table of both $C_{or}$ and $C_{T3}$ are also equal. Both tables have an OID attribute and all specialised attributes declared in that subclass, and hence both $S_{sub}$ are also the same.

Now, it is to prove that $C_{or} < C_{T2}$:

$$C_{or} < C_{T2},$$

$$\frac{R_{sub} \times S_{sub}}{\text{page size}} < \frac{R_{T2} \times S_{T2}}{\text{page size}},$$

$$R_{sub} \times (S_{oid} + S_{sub}) < \left( \sum_{i=1}^{m} R_i \right) \times \left( S_{oid} + S_{type} + \sum_{i=1}^{m} S_i \right),$$

$$\left( R_{sub} \times S_{oid} + R_{sub} \times S_{sub} \right) < \left( R_1 \times S_{oid} + R_1 \times S_{type} + R_1 \times S_1 \right)$$

$$+ \left( R_2 \times S_{oid} + R_2 \times S_{type} + R_2 \times S_2 \right)$$

$$+ \left( R_3 \times S_{oid} + R_3 \times S_{type} + R_3 \times S_3 \right)$$

$$+ \ldots + \left( R_m \times S_{oid} + R_m \times S_{type} + R_m \times S_m \right).$$  \hfill (12)
Since one of the subclasses in the right-hand side of Eq. (12) (e.g., \(R_2\) to \(R_m\) and \(S_2\) to \(S_m\)) is equal to \(R_{sub}\) and \(S_{sub}\), respectively, Eq. (12) is true. \(\square\)

**Example 2.** Using the same parameters as in Example 1, the costs are \(C_{T2} = (0 + 800 + 150 + 600) \times (0.4 + 0.15 + 0.12 + 0.14 + 0.01 + 0.01)/2 = 1550 \times (0.81 + (0.01 \times 2))/2 = 644\) I/O accesses, \(C_{T3} = C_{or} = 1550 \times (0.15 + 0.01)/2 = 124\) I/O accesses.

As anticipated, the performance of Type 3 and object-relational methods is much better than that of Type 2, because in the former, I/O accesses are isolated to the subclass only; in the latter the big table, which basically keeps all the records in the inheritance schema needs to be accessed.

### 4.4.3. Mixed queries

In practice, it is common that a query accesses both general and specialised attributes. This is a mixed type of superclass and subclass queries. **Mixed queries** are queries accessing general attributes declared in the superclass (superclass query) and specialised attributes in a subclass (subclass query). Mixed queries are common not only in two-level inheritance hierarchies, but also in multiple-level inheritance hierarchies (e.g., more than two levels). In multiple level inheritance hierarchies, there can be an arbitrary number of levels. For example, class \(A\) is a superclass of class \(B\) and class \(B\) is a superclass of class \(C\), and so on. A query on a class in the middle of the inheritance hierarchy (e.g., class \(B\)) can be regarded as a mixed query, as this class (e.g., class \(B\)) is a superclass of all subclasses (e.g., class \(C\)) below it, as well as a subclass of all of its superclasses (e.g., class \(A\)). For simplicity, in this paper we illustrate a mixed query using a two-level inheritance hierarchy example, which has been used in the previous sections.

**Query 3.** An example of a mixed query is as follows: “List the details of all commercial customers”. This query is similar to Query 2, but also accesses the superclass attribute, namely, commercial customer name.

The SQL statements for Type 2, Type 3 and object-relational (SQL\(_{T2}\), SQL\(_{T3}\) and SQL\(_{or}\), respectively) are as follows:

**SQL\(_{T2}\):**

```sql
Select Name, ACN
From CUSTOMER
Where CustType = ‘Commercial’;
```

**SQL\(_{T3}\) = SQL\(_{or}\):**

```sql
Select Name, ACN
From CUSTOMER, COMMERCIAL
Where CUSTOMER.ID = COMMERCIAL.ID;
```

Notice now that Type 3 and object-relational need a join between a subclass and its superclass, whereas Type 2 accesses the one table only.

Intuitively, join operation requires more costs than single table access. Therefore, the object-relational approach and Type 3 approach are more expensive that that of Type 2. Therefore, we can conclude that for mixed queries, Type 2 may outperform Type 3 and object-relational. Performance of Type 3 and object-relational in this sense is similar as shown by Example 2 previously
due to the small difference between the record size of a subclass table in Type 3 and in object-relational.

Type 2 however, suffers from several semantic, non-performance, problems. Type 2 will not work in the case of overlapping union inheritance, since there will be more than one type for a superclass object. Another problem with Type 2 is related to schema evolution whereby new subclasses are added to the inheritance schema. In this case, the table needs to be expanded every time a new subclass is added. Also, when the inheritance schema has a huge number of subclasses to form a long inheritance schema, it can be expected that the table will have excessive null values particularly in the specialised attributes.

4.4.4. Discussions

There are two issues in the evaluation of mapping of inheritance: one is the performance issue, and the other is the semantic issue.

The performance issue relates to how fast the query can access the tables generated by the transformation methodology. As join operations have been known to be one of the most expensive operations in RDB processing, we use a join operation as our indicator to evaluate the performance. For superclass queries, object-relational and Type 2 do not require any join, whereas Type 3 needs to join tables. Hence, Type 3 is inefficient. Lemma 1 has proved that for superclass queries, the object-relational model performs better than Type 2. In this regard, for superclass queries, the object-relational approach is the best. For subclass queries, none of the three methods requires any join. Based on the table size, Type 3 and object-relational are better than Type 2. For mixed queries, Type 3 and object-relational need join, whereas Type 2 does not.

The semantic issue relates to how the transformation methodology copes with different types of inheritance semantics. It has been mentioned that Type 2 has limited use as it does not support union inheritance. Therefore, Type 3 and object-relational are better.

Overall, comparing Type 3 and object-relational, both methods are equally good for subclass queries and mixed queries, but object-relational is better for superclass queries. We can conclude that having a “Type” attribute in the superclass table increases the efficiency in performance, and in many cases, like superclass queries, join operation can be avoided.

5. Performance evaluation of aggregation

5.1. Object-relational approach for mapping aggregation

Aggregation is basically the creation of a complex unit through an assembly of different parts. Aggregation combines low-level objects into composite objects and is often called a part-of relationship. To transform this aggregation type of relationship into relational tables, a virtual table called Aggregate table (or table AGG) is created. This table maintains the “part-of” relationships between the whole table and the part tables. By having one aggregate table, we avoid having separate relationships for each whole and part tables, and therefore we can fully represent the aggregation structure; i.e., between one whole table and one or more part tables. In the aggregate table only the relationships between the identifiers of the whole table and the part tables are stored. To maintain consistency in the aggregate table, the identifiers across different part
tables should be kept unique. If the number of the part tables is more than one, a new attribute "PartType" is used to distinguish the different types of the aggregate tables.

Consider the Lab–Computer–Printer–Scanner aggregation schema shown in Fig. 4(b) as an example. Using our object-relational transformation methodology, apart from the created original tables Lab, Computer, Printer and Scanner, an aggregation table AGG consisting of the keys of the whole and the part and a type attribute are also created. Suppose that the tables composition for the above aggregation hierarchy is shown in Fig. 15. The AGG table, which maintains the aggregation relationship, is illustrated in Fig. 16.

The PartType attribute of table AGG represents the type of the part classes (i.e., Computer, Printer and Scanner). In this example, the value of PartType attribute is the full string. It could have been encoded to, for example, one character instead of a full string, such as C for Computer, P for Printer and S for “Scanner”. The AGG table may not be in 3NF, but anomalies cannot occur, since this table is a table that stores only the relationships between the objects in the aggregation hierarchy. The data related to each of the objects is stored in the base tables. The PartType attribute in the AGG table is not supposed to be user-entered; instead it is derived directly from the base table names when an aggregation relationship is established.

5.2. Relational modelling approach for mapping aggregation

In relational modelling, there is no aggregation concept. Aggregation is often implemented as association. Using the same example: Lab–Computer–Printer–Scanner aggregation schema, a relational model using an ER diagram is shown in Fig. 17. The primary keys are underlined and the foreign keys are indicated with broken underline.

Since the cardinality of each relationship is one-to-many, the size of the part tables (i.e., Computer, Printer, Scanner) is added with one more attribute; that is, an FK from the whole table (i.e., Lab) PK.

<table>
<thead>
<tr>
<th>Lab</th>
<th>Computer</th>
<th>Scanner</th>
<th>Printer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Room</td>
<td>ID</td>
<td>Brand</td>
</tr>
<tr>
<td>1</td>
<td>BG121</td>
<td>1</td>
<td>HP</td>
</tr>
</tbody>
</table>

Fig. 15. Sample data of an aggregation schema.

<table>
<thead>
<tr>
<th>AGG</th>
</tr>
</thead>
<tbody>
<tr>
<td>WholeID</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 16. An aggregate table.
5.3. Analysis

In analysing our object-relational transformation method, we compare it with the aggregation implementation using an association approach (conventional approach).

5.3.1. Aggregation versus association

As mentioned earlier, there are four kinds of aggregation, in particular: existence dependent/independent aggregation, ordered aggregation, exclusive/non-exclusive aggregation and homogeneous aggregation.

Existence dependent/independent aggregation. Fig. 18 shows a comparison between the implementation of an existence-dependent aggregation using our object-relational approach in which aggregation is implemented as an aggregation, and an implementation using an association model. In the example, our object-relational approach enforces that the deletion of a Building will cause the deletion of that particular Building record in the AGG table. The constraint enforces that Doors/Walls/Windows that do not appear in the AGG table to be deleted as well. However, this can also be achieved by the association model by specifying that the referential integrity constraint of the FK is delete cascade.

Another point is that in an existence-dependent aggregation, there will not be any part that does not belong to any whole object. In the object-relational approach, this is enforced by the constraint that if the new part ID does not exist in the AGG table, such part of the record cannot
be inserted. Using the association approach, we can achieve this by specifying that the FK be NOT NULL.

Hence, for existence dependent aggregation, there is no major difference between the two approaches. However, later we will explain that other aggregation models are very specific and can be implemented by our object-relational approach only.

In an existence independent aggregation, the existence of the part is independent. For example (see Fig. 19), if for some reasons a Lab is removed, the Computer, Printer and Scanner still exist. Using the object-relational approach this can be done by removing the delete restriction in each of the part tables. Using the association approach, this is realised by removing the NOT NULL constraint in the FK.

Again, for the existence independent aggregation, there is no major difference between the two approaches in achieving object integrity and consistency.

**Ordered aggregation.** In an ordered aggregation, the order of occurrence of the part objects in the composition is significant to the model. In contrast, ordering in the association model is missing. Association is a mere connection between object types, and hence there is no direct way to maintain the order of all associated objects of different types. Fig. 20 shows an example whereby ordering is maintained in the aggregation model, but absent in the association model.

**Exclusive/non-exclusive aggregation.** In the example shown in Fig. 21, we need to ensure that every part object is exclusively owned by a particular whole only. The object-relational approach has the ability to model the situation. The AGG table contains PartNo (can be EngNo or BodyNo) as the PK. This ensures that every part object appears only once in the AGG table, which also implies that it can be related to only one particular whole object.

On the other hand, the conventional approach using an association model does not have the ability to capture the exclusiveness in semantics. We can easily have several Cars with the same EngNo or BodyNo, which violates the exclusiveness.
In a non-exclusive composition, a part of one whole object may be shared or referenced by other whole objects, and thus the part is not exclusive. For example, a BinaryFile or a TextFile can be referenced by more than one Directory (see Fig. 22). This can be implemented by the object-relational approach as long as the PartNo in the AGG table is either a BinaryFile or a TextFile. Using the association model, the non-exclusiveness is implemented through a many-to-many relationship. Hence, there is no real difference between the two approaches in implementing non-exclusive aggregation.

**Homogeneous aggregation.** Fig. 23 gives a comparison of implementation of a homogeneous aggregation using the object-relational approach and the conventional approach. The main advantage of modelling the homogeneous aggregation using our object-relational approach is that the model is flexible enough for further extension or modifications to include components of another type. In the case of a mixture of homogeneous and heterogeneous components, the homogeneous composition is indicated by the cardinality, namely, one-to-many.

### 5.3.2. Quantitative analysis

Aggregate queries are normally categorised into scalar aggregates and aggregate functions [11]. Scalar aggregate queries produce single values for a given set of records (i.e., table), whereas
aggregate function queries generate a set of values for a given table. The former is like grouping the whole table and produces a single value, whereas the latter is like grouping the table into several groups, and for each group a single value is produced. In the performance evaluation of aggregation mapping, we employ both aggregate queries as our benchmark.

**Query 4.** An example of a scalar aggregate query is as follows: “Retrieve the total number of computers in Lab BG121”.

The SQL statements using an object-relational approach (SQL\textsubscript{or}) and conventional approach in which aggregation is implemented as an association (SQL\textsubscript{assoc}) are as follows:

SQL\textsubscript{or}:

```sql
Select Count (*)
From AGG, LAB
Where AGG.WholeID = LAB.ID
AND AGG.PartType = ‘‘Computer’’
AND LAB.Room = ‘‘BG121’’;
```

SQL\textsubscript{assoc}:

```sql
Select Count (*)
From COMPUTER, LAB
Where COMPUTER.LabID = LAB.ID
AND LAB.Room = ‘‘BG121’’;
```

Using either the object-relational model or the conventional (association) model, a join operation is necessary between a whole table and another table (in the case of object-relational the second table is the AGG table, whereas in the association model the second table is the part table itself).

Comparing the efficiency of the two join operations depends very much on the second table in the join since the first table involved in the join, namely, the whole table is the same for both methods. Hence, we compare the table size of the second tables, namely, the AGG table (for the object-relational approach) and the Computer table (for the association approach).

Using the object-relational approach, the record size and the number of records of table AGG are as follows:

\[
S_{\text{AGG}} = S_{\text oid whole} + S_{\text oid part} + S_{\text type} \quad \text{and} \quad R_{\text{AGG}} = \sum_{i=2}^{m} R_i. \tag{13}
\]

The record size of the AGG table \(S_{\text{agg}}\) is the sum of the size of the three attributes. The number of records of the AGG table is the sum of all records from the part tables. Using the sample data in Fig. 16, the number of records of the AGG table is \(R_{\text{agg}} = R_2 + R_3 + R_4 = 4 + 1 + 2 = 7\) records \((R_2\) is the number of computer objects, \(R_3\) is the number of scanner objects, and \(R_4\) is the number of printer objects). Suppose that each of the three attributes of the AGG table is only 0.01 Kb, total record size is only 0.03 Kb. Overall, the AGG table occupies 7 records \(\times\) 0.03 Kb = 0.21 Kb.

Using the association approach, the record size of the part table is then determined by

\[
S_{\text{part}} = S + S_{\text FK}. \tag{14}
\]
The original record size of each part class is denoted by \( S \) (the same as \( S_{\text{ap}} \) or \( S_{\text{ab}} \) as in Eq. (3)). The number of records of each part table is \( R_i \), where \( i \) indicates the \( i \)th part class. Using the same example as the above, the number of records of table Computer is 4 records. Assuming that the total size of all attributes in class Computer is 0.8 Kb, the record size of table Computer, including the FK becomes \( S = 0.8 \text{ Kb} + 0.01 \text{ Kb} = 0.81 \text{ Kb} \).

Comparing the two figures, in this query example, the AGG table is smaller than the Computer table. However, if the query is to retrieve the number of scanners in a particular lab, since the scanner table is very small, the association approach will deliver better performance. Moreover, if the query is to also retrieve the details of a part table, such as “retrieve the details of all computers in lab BG121”, the object-relational approach needs to join three tables, namely, the whole table (e.g., table Lab), the AGG table, and a part table (e.g., table Computer). On the other hand, the conventional approach needs to join two tables only: the whole table (e.g., table Lab) and a part table (e.g., table Computer). Without further proof, we can say that the conventional approach can deliver better performance.

**Query 5.** An example of an aggregate function query is as follows: “Retrieve total number of each part (e.g., Computers, Scanners and Printers) placed in Lab BG121”.

In our object-relational methodology, a join between the AGG table and the whole table is necessary. The SQL statement (SQL\textsubscript{or}) is as follows:

```sql
SQL_{\text{or}}:
Select PartType, Count (*)
From AGG, LAB
Where AGG.WholeID = LAB.ID
AND LAB.Room = 'BG121';
```

In contrast, using the association approach, to retrieve all parts of a composite table, separate queries consisting of a join operation between the whole and one part tables have to be performed on each of the part table. The SQL statement (SQL\textsubscript{assoc}) is as follows:

```sql
SQL_{\text{assoc}}:
Select 'Computer', Count (*)
From COMPUTER, LAB
Where COMPUTER.LabID = LAB.ID
AND LAB.Room = 'BG121'
UNION
Select 'Scanner', Count (*)
From SCANNER, LAB
Where SCANNER.LabID = LAB.ID
AND LAB.Room = 'BG121'
UNION
Select 'Printer', Count (*)
From PRINTER, LAB
Where PRINTER.LabID = LAB.ID
AND LAB.Room = 'BG121';
```
As shown by SQL_{assoc}, each subquery performs a join between a part table and the whole table. On the other hand, in SQL_{or}, the join is only between the AGG table and the whole table. Assuming that the size of the AGG table is approximately equal to the average part table size, the performance of the conventional approach is downgraded by extra join operations. The object-relational approach needs one join operation whereas the conventional approach needs \( m - 1 \) joins where \( m - 1 \) is the total number of part tables.

However, as discussed previously in the scalar aggregate query example, if the query needs to retrieve the details of each part table, using the object-relational Approach, additional join operations with each of the part table, as well as the AGG table, become obligatory, and as a result, additional overhead is required.

5.4. Discussions

Performance in terms of query execution of the object-relational approach may not offer better performance compared with the conventional approach using an association due to the additional table: the AGG table. Consequently, join operations, which are often required by the query may involve this additional aggregate table, as well as the whole and the part tables.

However, the object-relational approach for mapping aggregation offers advantages, which can be summarised as follows.

5.4.1. Support of model extension

The use of an Aggregate table (i.e., table AGG) to maintain the relationships between the whole objects and its part objects in the object-relational model demonstrates the ability of this technique to support model extension. When a new part table is added to the schema, there will be no modifications needed in any of the existing tables. For every record of the new table, a record consisting of the whole number, the new part number, and the new table name is inserted into the Aggregate table. This rule is applicable to any of the types of aggregation mentioned above.

Consider the homogeneous aggregate of Fig. 4(f) as an example. Assume that the model is extended into a mixture of homogeneous and heterogeneous aggregation by adding a Commercial class to the DailyProgram (see Fig. 24). Fig. 24 shows that the aggregation model needs one extra table that independently stores the new part details. In contrast, the conventional approach using

---

![Aggregation model](image1)

![Association model](image2)

Fig. 24. Homogeneous aggregation extension of Fig. 4(f).
an association model needs two additional tables: one for the new part and the other to store the new association relationship between the whole and the new part table.

5.4.2. Reusable access methods

The object-relational approach supports reusable methods to access the tables. Any extension in the aggregation model will not cause any major modifications in the access methods. Using the example shown previously in Fig. 24 where DailyProgram has been added with one more part called Commercial, to access all parts of a particular DailyProgram in the aggregation model, the same access method can still be applied:

```
Select PartNo, PartType
From AGG
Where DProgNo = ‘...’;
```

Even if the model is further extended or modified in the future, the access method remains the same. This is mainly because in the object-relational model, access is only to the Aggregate table that maintains the relationship between the whole and the parts. On the contrary, in the conventional approach using an association model, every association path has to be traced in order to get the whole information about the aggregation relationship. To access all parts of a particular DailyProgram, two different accesses need to be carried out.

```
(i)
Select CommNo
From COMMERCIAL, SPONSORED BY
Where CommNo = SPONSORED BY. CommNo
AND SPONSORED BY. DProgNo = ‘...’;
(ii)
```

If the association model is further extended by adding another new part table, the number of access methods will also be increased.

6. Performance evaluation of association

For the performance evaluation of mapping association, we start by describing the relational approach first, followed by our object-relational transformation approach. In this way we are able to show that our object-relational methodology is a superset of the conventional relational modelling, since the object-relational methodology includes lists/arrays and bags, as well as the traditional sets.

6.1. Relational modelling approach for mapping association

Association relationships are often related to sets. Relational data structures can be related to the concepts of sets through the fact that records are not in any particular order, and duplicate records are not allowed. Therefore, transformation of association relationships with a set semantic into relational tables is identical to the well-known transformation of many-to-many or one-to-many relationships from relational modelling to relational tables.
In relational modelling, many-to-many relationships are converted into tables in which the PK is a composite key obtained from the participating entities. Should there be any attributes of the relationships, these will automatically be added to the tables that represent the many-to-many relationships. Likewise in object modelling, if a class has a set relationship with another class and the inverse relationship is also a set, the transformation of such association relationships is identical with the many-to-many relationships transformation from relational modelling to relational tables where a table is created to represent the set relationship. This transformation strategy also enforces that each element within a set cannot be duplicated which is realised by the implementation of the composite PK of the relationship tables.

In one-to-many relationships, as in the relational modelling, the PK of the one-side is copied to the many-side to become an FK. In other words, there is no special treatment necessary for the transformation of association relationships having a set semantic.

6.2. Object-relational approach for mapping association

Association relationships in object-orientation are not only of type set, and other collection types are available, such as list, array and bag. They differ from sets in duplication and ordering. These make the transformation of association relationships more complicated than that of sets. Conventional transformation of association relationships of type set cannot be applied to lists/arrays and bags.

Our previous work [23] proposes a simple transformation procedure for collection types (including lists/arrays and bags). The class that comprises a collection of other associated class is called the parent class, whereas the class that forms the collection is called child class. For example, in the relationships of class Program having a list of Commercials (see Fig. 25 as an example), class Program is a parent class, whereas class Commercial is a child class. This naming convention is used in the context of association transformation of type collections (i.e., lists/arrays and bags) only. It is used to make class name referral easier.

The proposed transformation procedure is as follows. Firstly, create a table for each class. The parent class will be converted to a table, and so will the child class. Secondly, create a table to represent the collection association relationship. The attributes of this table are the PK of the parent class, the index, and the key of the child class. The PK is a composite attribute of ParentID and Index:

\[
\text{RelationshipTable}(\text{ParentID, Index, ChildID})
\]

The attribute Index is used to show the index of each element within one collection. The value of attribute Index can be system-provided or user-specified. In the case where the relationship type

![Fig. 25. Association schema.](image-url)
is either a list or an array, the values of attribute Index represent the order of those elements within one collection. In this way, element ordering within a list or an array is maintained.

However, for bags where the elements’ order is unimportant, attribute Index is still needed to accommodate the duplication feature of bags. To explain how this is accomplished, we will explain the opposite case that is if the attribute Index were not used. Without attribute Index, the PK of the RelationshipTable has to be a composite PK between ParentID and ChildID. This then becomes the same as the existing treatment of many-to-many relationships in relational modeling. Using such a composite PK, there is no way that the value of the pair of ParentID and ChildID can be duplicated, as these two attributes form a composite PK.

With attribute Index, the above duplication problem in bags is solved. When ParentID and ChildID records are duplicated, the value of attribute Index is kept distinct for each pair, such as attribute Index value 1 for the first pair of ParentID and ChildID and attribute Index value 2 for the duplicated pair. This is the main reason why attribute Index is still used for bags. It is not to maintain the order of the elements within a bag, but to allow duplicate elements in a bag to exist.

6.3. Analysis

Since the existing RDB design does not explicitly include collection types as proposed by ODMG, a fair comparison between the existing RDB design methodology and our object-relational transformation methodology cannot be presented. Nevertheless, we still make some comparisons. Consider the association schema in Fig. 25 (in the previous section) as an example. The schema shows an association between Program and Commercial, where a Program usually has more than one type of Commercial. Furthermore, a particular commercial can be repeated several times during the telecast of a program. The system must be able to keep track of the order of the commercials. A list is the most suitable type of data collection in this kind of situation, since it allows repetition of the items and still keeps track of the order of items. As a comparison, consider the following two queries: one query on the order of elements within a collection, and the other query on the issues of element duplication within a collection.

6.3.1. Order element query

An “Order Element” query is an association query, which exposes the fact that elements within a collection must be in a certain order.

**Query 6.** The following query involves a list: “Return the first commercial that appears during the showing of Program P1”. This query emphasises the order of commercials, from which we would like to obtain the first commercial.

The tables produced by the object-relational transformation rules are:

Program(ProgramID, ...)
Commercial(CommercialID, ...)
Lists(Index, CommercialID, ProgramID)

**SQL**:

```sql
SELECT Index, CommercialID
FROM Lists
WHERE ProgramID = 'P1'
AND Index = 1
```
The existing RDBs do not support the concept of List collection types. The relation between Program and Commercial is treated as an ordinary many-to-many relationship. Consequently, there is no way to implement query 6, because many-to-many relationships do not keep track of the order of each collection. A common transformation of many-to-many relationships as described in many database textbooks (e.g., [9]) is to create a new table consisting of the PK of each table participating in the relationship as a composite PK, and any other non-key attributes relevant to the relationship. There is no mention of ordering of any kind, even though practitioners may resolve this problem by introducing an additional attribute to reflect the timestamp. However, due to unnecessary functional dependencies, this may introduce anomalies according to the relational theory.

6.3.2. Duplicate element query

A “duplicate element” query is an association query, which exposes the fact that elements within a collection can be duplicated.

Query 7. The following query involves a bag: “Display a bag of Ratings acquired by Movie P3”. This query emphasises that ratings can be duplicated for each movie.

The transformed tables using object-relational are as follows:

\[
\begin{align*}
Movie & (ProgramID, Title, ...) \\
Bag & (BagNo, ProgramID, Rating)
\end{align*}
\]

\[
\text{SQL} \text{or:} \\
\text{SELECT} \text{BagNo, Rating} \\
\text{FROM Bag} \\
\text{WHERE ProgramID = 'P3'}
\]

Ratings in Movie are usually implemented as a multivalued attribute in ER. This schema can be translated to a Rating table with a composite PK of Rating and the PK of Movie. However, this composite PK does not allow any repetition. Hence, the above query cannot be implemented directly in RDBs if we use multivalued attributes in ER.

7. Related work on object-relational transformation methodology

There have been several different approaches to bringing OO and relational concepts together by developing transformation rules from an OO conceptual model into a relational logical model. However, most of them are done in a quite different context and with different goals from those that we proposed, because they mainly address the problem of the development of data model translators in heterogeneous distributed database systems [13,18,26].

Other works, e.g., [5], emphasise a special type of application (i.e., Real-Time DBMS), a relational DBMS built for high performance transaction processing which utilises the OO paradigm in order to extend the classical viewpoint of the relational DBMS. This work does not actually describe the details of the schema transformation compiler procedures. In contrast, our object-relational methodology is not application-specific; it encompasses the most important aspects of object-orientation and the implementation of those aspects in any current relational DBMS systems.
Similar approaches to ours have been adopted by Rumbaugh et al. [27] and Kroenke [17], who developed rules for transforming OO conceptual model to relational tables. However, the techniques are described in a general manner without any emphasis on maintaining the semantic concepts of object-orientation in the relational tables. The transformation procedures discussed in our work encompass many different structures and aspects of object-orientation, such as different types of Inheritance structures (union, mutual exclusive, partition and multiple inheritance), Aggregation (whole/part) structures, Non-hierarchical structures, as well as collection types of attributes (set, array, list and bag). Moreover, we go one step further by evaluating the performance of relational implementation of our methodology by comparing it with that of conventional RDB design. This performance evaluation is important as it proves the efficiency of our methodology.

8. Conclusions and future work

Our goal as stated at the beginning of this paper, is to verify the efficiency of the operations on the relational tables derived using our object-relational transformation methodology. We achieved this goal by comparing the performance of object-relational transformation methodology with that of the conventional relational method. We have evaluated the mapping of inheritance, aggregation and association hierarchies of the object model.

From the results that we have gathered, the implementation using our object-relational transformation has proven its superiority over the implementation that uses the relational method, especially for inheritance superclass and subclass queries in which join operation is avoided.

For the aggregation hierarchies, we discussed how our object-relational model offers better solutions through the use of a separate aggregate table than the conventional approach using an association model.

We have also described how in the case of collection types, particularly queries 6 and 7, conventional relational modelling is not able to capture these features in the design stage and, consequently, is not able to be implemented correctly. The object-relational methodology proves that it is able to not only model collection types but also implement them efficiently without the loss of its semantics.

Based on the analysis of the sample queries above, it is obvious that the additional concepts in the transformation methodology have contributed to the performance of the operations. Our future work is to improve the transformation methodology by incorporating the constraints and dynamic parts of object-orientation.

References


Johanna Wenny Rahayu received a Ph.D. in Computer Science from La Trobe University, Australia, in 2000. Her thesis, supervised by Profs. Elizabeth Chang and Tharam Dillon, was in the area of object-relational database design and transformation methodology. This thesis has been awarded the 2001 Computer Science Association Australia Best Ph.D. Thesis Award. Dr. Rahayu is currently a Senior Lecturer at La Trobe University. She has published two books and numerous research articles.
Elizabeth Chang is Associate Professor in the Department of Computer Science at Newcastle University, Australia. Her research interests are in the areas of User Interface Development, Software Engineering, and Databases. She has been appointed as a technical program committee member and keynote speaker in major international conferences, and published numerous papers in journals and international conferences.

Tharam S. Dillon is Professor of Computer Science at La Trobe University in Melbourne, Australia, and Director of the Applied Computing Research Institute. He has carried out research in the software engineering, database, and knowledge-based systems areas, and, in particular, he has worked on object-oriented approaches to each of these areas. He has also worked with industry and commerce in developing systems for banking and finance, power systems, telecommunications, and health care systems; and is editor-in-chief of the International Journal of Computer Systems Science Engineering.

David Taniar received his Ph.D. in Computer Science from Victoria University, Australia, in 1997 under the supervision of Professor Clement Leung. He is currently a Senior Lecturer at the School of Business Systems, Monash University, Australia. His research interests are in the areas of Applications of High Performance Computing in Databases, Data Warehousing, and Data Mining. He has published four computing books, and numerous research articles. He is also a Fellow of the Royal Society of Arts, Manufactures and Commerce.