Pergamon

Information Systems Vol. 19, No. 4, pp. 33–54, 1994

0306-4379(94)E0004-9
0306-4379/94 $7.00 + 0.00

A NEW METHOD FOR ESTIMATING THE NUMBER OF OBJECTS SATISFYING AN OBJECT-ORIENTED QUERY INVOLVING PARTIAL PARTICIPATION OF CLASSES

Wan-Sup Cho[1], Chong-Mok Park[1], Kyu-Young Whang[1], and Sang-Hyuk Son[2]

[1]Department of Computer Science, Korea Advanced Institute of Science and Technology, Taejon, 305-701, Korea
[2]Department of Computer Science, University of Virginia, Charlottesville, VA 22903, USA

(Received 8 February 1995; in final revised form 15 September 1995)

Abstract — The intermediate result cardinality – the number of objects satisfying a condition given in a query – is an important factor for estimating the cost of the query in query optimization. In this paper we show that an object-oriented query often involves partial participation of classes in a relationship. We then present a new technique for estimating the intermediate result cardinality in such a query. Partial participation has not been considered seriously in existing techniques. Since the proposed technique uses detailed statistics to accommodate partial participation, it estimates the intermediate result cardinality more accurately than existing ones. We also show that these statistics are easily obtained by using inherent properties of object-oriented databases.

Key words: object-oriented databases, query optimization, cost model, selectivities, intermediate results

1. INTRODUCTION

Object-oriented database management systems (OODBMSs) are adequate for supporting new database applications such as engineering databases and multimedia databases [1, 12, 17]. Although there are a number of OODBMSs, most of the current OODBMSs do not provide sufficient support for the database management facilities [13, 17], such as high-level query languages, query optimization, dynamic schema changes, indexing techniques, concurrency control, and recovery. Among these facilities, query optimization is essential for efficient processing of high-level queries.

In query optimization, the system automatically generates a set of reasonable execution plans for processing a given query, and selects the one with the minimum estimated cost. One of the most important factors that affects the cost estimation of an execution plan is the *intermediate result cardinality* – the number of objects in an intermediate result [3, 7, 19, 21]. This is because the intermediate result becomes the operand of the next operation in the execution plan, and the cost of the next operation can be estimated mainly based on the number of objects in its operands [18, 19, 21]. Hence, accurate estimation of intermediate result cardinality is a prerequisite for good query optimization.

Intermediate results cardinalities are estimated by multiplying the cardinalities of unconditional joins† [24] with the selectivity factors of conditions [19, 21]. The *selectivity factor* of a condition represents the portion of objects that satisfy the condition. In this paper, we propose new techniques for estimating the cardinalities of unconditional joins, selectivity factors, and intermediate results cardinalities in object-oriented queries.

We first show that an object-oriented query often involves partial participation of classes in a relationship and compare the case with that for relational queries. *Partial participation* means that only a proper subset of objects in a class are related to the objects of another class [6]. Two objects are *related* (or *has a relationship*) if an attribute in one object has the Oid of another object as its value. For example, if only a subset of objects in the class Student are related to the objects in the class Car via the relationship own, then the class Student partially participates in the relationship own. Here, we call the class Student a *partial participation class* with respect to the relationship own.

†A join without selection conditions on either relation.

We then propose new techniques for estimating the cardinalities of unconditional joins and selectivity factors of a query involving partial participation classes. Partial participation has not been considered seriously in the literature [15]. Most of existing query optimization techniques [2, 8, 11, 16, 19] except for Whang et al.[24, 21, 22] have not considered partial participation. We discuss in Section 4.1 that these conventional techniques often incur large estimation errors for the queries involving partial participation classes. We also consider the effect of multi-valued attributes on the estimation of intermediate results cardinalities. Most of existing estimation techniques [2, 16, 19, 24, 21] have not considered multi-valued attributes seriously. The proposed technique for estimating the cardinalities of unconditional joins extends Whang's technique [24, 21, 22] so as to consider the characteristics of object-oriented databases. The proposed techniques require new types of statistics for estimating query costs. We also show that these statistics can be easily obtained by taking advantage of inherent properties of object-oriented databases.

The paper is organized as follows. In Section 2, we briefly review the basic concepts of object-oriented databases necessary for discussing query optimization. In Section 3, we explain the reasons why partial participation occurs frequently in object-oriented queries and compare the case with that for relational queries. In Section 4, we discuss the impact of partial participation and multi-valued attributes on the estimation of the intermediate results cardinalities and present new estimation techniques. We then compare our techniques with the conventional ones to show the significance of the role of partial participation. Finally, we conclude the paper in Section 5.

2. BACKGROUND

In this paper, we use a generally accepted subset of object-oriented data models described in the literature [1, 12]. Objects are uniquely identified by *object identifiers* (*Oids*). Objects having the same properties (attributes and methods) are classified into a *class*. The definition of a class forms a two-dimensional rooted directed graph of classes called the *schema graph* for that class [12].

Fig. 1 is an example schema graph for the class Person. The class Person is the root of the *aggregation graph* that includes the classes Vehicle and Company. The class Person is also the root of the *inheritance graph* (or *class hierarchy*) involving the classes Employee and Student. For any

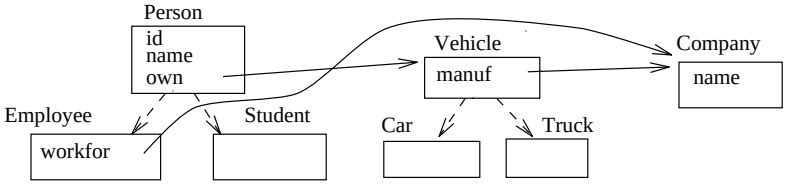

Fig. 1: Schema Graph for the Person, Vehicle, and Company classes.

class C, C^* denotes all the classes in the inheritance graph rooted at the class C. For example, Person* denotes the (union of the) classes Person, Employee, and Student, which belong to the inheritance graph rooted at the class Person. All the properties of a superclass are inherited to its subclasses.

An attribute whose domain is a class in a schema graph represents a relationship between the class in which the attribute is defined and the class that is the domain of the attribute. For example, the attribute own represents the relationship between the classes Person and Vehicle. The domain of an attribute may be the union of a given class and all its subclasses [12]. For example, in Fig. 1, the domain of the attribute own is the entire inheritance graph rooted at Vehicle, i.e., Vehicle*, and the values of the attribute own are the Oids referring to the objects of the class Vehicle or any subclass of it. If a class is the domain of more than one attribute, we call the class a *common domain* for the attributes. For example, in Fig. 1, the class Company is the common domain of the two attributes manuf and workfor. If an attribute has a single value for a particular object, we call it a *single-valued attribute*; otherwise, i.e., if it may have more than one value, we call it a *multi-valued attribute*.

Most of the object-oriented query languages [10, 12] are extensions of relational ones supporting object-oriented concepts [1, 12]. In this paper, we use the query model proposed by Kifer et al.[10]. Concentrating on the nested predicates in which methods and views are excluded. Although this kind of query is a small part of Kifer's query model, it is an important class of queries that needs careful investigation on the costs for effective query optimization because nested predicates involve multiple expensive joins.

Example 1 In Fig. 1 consider the following query for finding the names of all persons who own a truck manufactured by "Ford". Here we assume that the attribute own is a single-valued attribute.

$SELECT$	p.name
$FROM$	Person p
$WHERE$	p.own[Truck].manuf.name = "Ford";

An important feature of Kifer's query model is the *extended path expression* (or simply the *path*) [10] which has the following form

$$P \quad = \quad C_0.A_1\{[C_1]\}.A_2\{[C_2]\}...A_n\{[C_n]\}, \quad n \geq 0 \tag{1}$$

where $n \geq 0$, $C_0, ..., C_n$ are class names, $A_1, ..., A_n$ are attribute names, and braces denote optional terms (i.e., only C_0 is mandatory), If "$[C_i]$" is omitted, we assume that the domain of A_i is the class (or inheritance graph) that is defined in the schema as the domain of A_i. The *length* of the path P is defined as the number of attributes, n, in P. If the length of a path is zero, it represents just a class. We call the first class C_0 the *starting class* and the last attribute A_n the *ending attribute* of the path P [2].

In the path P, we call C_{i-1} the *codomain* and C_i the *domain* of the attribute A_i. Note that users can limit the domain of A_i to a specific class (or its inheritance graph) by designating the class in the brace followed by A_i. We call this designation *domain substitution* of A_i. If the domain of A_i is substituted by the class C_i, the codomain of A_{i+1} is automatically limited to (or substituted by) C_i. We call this limitation *codomain substitution* of A_{i+1}. As an exception, codomain substitution for the first attribute A_1 occurs when the starting class is designated to be a specific class rather than the union of all the classes in an inheritance graph. For example, in the path Student.own[Truck].color, the codomain for the attribute own is limited to the class Student, and the domain to the class Truck, i.e., we have both codomain and domain substitutions.

The path is a convenient and natural way of expressing queries along the aggregation graph [10, 12]. Furthermore, domain and codomain substitutions in a path allow users to express more delicate queries [10, 12].

Example 2 In Fig. 1, consider a path Person*.own.manuf.name. We can substitute the codomain and domain classes of the attribute own in the path in various ways as follows. For simplicity, we abbreviate the class names Person, Employee, Student, Vehicle, Truck, and Car as P, E, S, V, T, and C.

P_1 : P.own[V*].manuf.name	P_5 : E.own[V*].manuf.name	P_9 : S.own[V*].manuf.name
P_2 : P.own[V].manuf.name	P_6 : E.own[V].manuf.name	P_{10} : S.own[V].manuf.name
P_3 : P.own[C].manuf.name	P_7 : E.own[C].manuf.name	P_{11} : S.own[C].manuf.name
P_4 : P.own[T].manuf.name	P_8 : E.own[T].manuf.name	P_{12} : S.own[T].manuf.name

The path P_1 denotes the manufacturer's names of vehicles (including trucks and cars) owned by persons. Similarly, path P_{12} denotes the manufacturer's names of trucks owned by students. The rest can be explained similarly.

In a path, each attribute whose domain is a class implicitly specifies a join between codomain and domain classes of the attribute. We call this kind of join *implicit join* [8, 10, 12, 16]. For example, two implicit joins are specified in the path Person.own[Vehicle].manuf.name: (1) one between Person and Vehicle via the attribute own, and (2) another between Vehicle and Company via the attribute manuf. All joins appearing in the remainder of this paper are implicit joins.

In Eq. (1), we call a sequence of $n+1$ objects $[o_0, o_1, ..., o_n]$ a *path instance* of the path P if it satisfies all of the following conditions [2, 10]: (1) o_0 is an object (or instance) of the class C_0, and (2) o_i $(0 < i \leq n)$ is the value of the attribute A_{i-1} of object o_{i-1}. For example, the sequence of objects [p, t, c, "Jones"] is a path instance for Person.own[Truck].manuf.name, where p is a Person object, p.own has the Oid of the Truck object t, t.manuf has the Oid of the Company object c, and c.name is the string "Jones".

We use the following statistics reflecting the state of the database for the path P. We present an efficient method for obtaining these statistics in Sections 4.2 and 4.3.

- $n(P)$: the number of path instances in the path P. If the length of the path P is zero, $n(P)$ is the number of objects of the class specified by P. For example, n(Student) is just the number of Student objects. On the other hand, n(Student.own[Car]) is the number of pairs of objects (i.e., path instances) [s, c], where s is an object in Student, c in Car, and s.own contains the Oid of c.

- $dv(P)$: the number of distinct values of the ending attribute of the path P in the path instances. For example, dv(Student.name) is the number of distinct name in the path instances for Student.name, and dv(Student.own[Truck].manuf) is the number of distinct Company objects in the path instances for Student.own[Truck].manuf.

We make the following assumptions.

- For considering codomain/domain substitution in an object-oriented query, we extend the definition of the uniform distribution [4, 19] of an attribute as follows: If the domain of a complex attribute A_i consists of $D_1, ..., D_n, n > 1$, then the Oids of individual D_k's that appear in A_i of C_i are uniformly distributed in the class $C_i, i = 1, 2, ...$ For example, in Fig. 1, the Oids of Car that appear in the attribute own are uniformly distributed in each of Person, Employee, and Student classes. Similarly, the Oids of Vehicle and Truck are also uniformly distributed in each of these classes. Note that this assumption does not modify the commonly accepted definition of uniform distribution [4, 19], but extends the definition according to the subclasses in the class hierarchy. If $n = 1$, i.e., the inheritance graph consists of a single class, then this assumption is reduced to the conventional definition [4, 19].

- An Oid consists of a <*class-id, instance-id*> pair, where *class-id* is the identifier of the class to which the object belongs, and *instance-id* is the identifier of an object (instance) either within the class or within the entire database [12]. This kind of Oid structure is used in ORION [12].

3. PARTIAL PARTICIPATION OF CLASSES

In this section we show that an object-oriented query often involves partial participation of classes in a relationship and compare the case with that for relational queries. We also discuss the effect of partial participation on the estimation of the cardinalities for unconditional joins.

A relational join is typically done between a foreign key attribute (FKA) and a primary key attribute (PKA); here, both relations of the join may be partial participation relations. We call such a join a *partial participation join*. If the FKA allows null values, then the tuples having null values for the FKA do not participate in the join. Thus, the relations including the FKA partially participates in the join. Likewise, if only a subset of values of the PKA is referred to by the values of the FKA, then the relation including the PKA partially participates in the join. Tuples that do not participate in the join are called *dangling tuples* [6, 15]. For example, consider a join between Car and Person where Car.owner is a foreign key corresponding to the primary key Person.id. Then, the Car tuples that have null values in Car.owner (i.e., cars with no owner) are dangling tuples. Likewise, the Person tuples that are not referred to by Car.owner (i.e., persons having no cars) are also dangling tuples. Much of the work on relational query optimization ignores the effect of partial participation joins [15, 18, 19]. This is because it is assumed that referential integrity generally holds and the tuples of the FKA-side relation totally participate in the join. However,

as indicated by Korth and Silberschatz [15], if *both* relations have dangling tuples, the selectivity factor produced by Selinger's method [19] is higher than the correct value. As a correction factor for partial participation joins, Whang et al. [24, 21, 22] introduced the concept of *join participation ratio* (defined originally in [24] as *join selectivity*). We explain the concept in Section 4.1 in detail.

Another source of partial participation joins in relational databases is the appropriate use of selection and join operations. After selection operations are performed, a join becomes a partial participation join because only the qualified objects participate in the join. We call this kind of partial participation *partial participation by selections*.

There is an important difference between partial participation by selections in relational databases and the partial participation in object-oriented databases. In partial participation by selections, since the proportion of participating objects to the entire objects depends on the selectivity factor of the selection operations, it is difficult to reflect the effect of partial participation in query optimization. We need to handle this effect via either assuming attribute independence [18, 19] or maintaining a special data structure for keeping the joint distribution of selection and join attributes [18]. However, it has been argued that attribute independence leads to the upper bound of the actual expected cost [4] and maintaining special data structures may be too expensive [18]. On the other hand, partial participation in object-oriented databases is visible through existence of subclasses, which corresponds to the results of selection operations in relational databases, and domain/codomain substitutions in queries. Since subclasses are modelled at the schema level, we can easily obtain the statistics needed to account for partial participation from these subclasses and maintain them in the system catalog. The detailed method for obtaining these statistics is fully discussed in Section 4.2 and Section 4.3.

In object-oriented queries, partial participation joins occur frequently. These are mainly caused by codomain and/or domain substitutions and common domains. We use Fig. 2 as an example. It consists of two class hierarchies rooted at the classes C and D. The attribute A_i is defined for the class C and is inherited to the subclasses. The domain of A_i is the entire class hierarchy rooted at the class D (i.e., D^*). The grey-colored subregions of C_i and D_j denote the objects participating in the join $C_i.A_i[D_j]$, and the rest denote those not participating in the join. We identify three cases where partial participation occurs by using the schema graph in Fig. 2.

The first case occurs when the domain of an attribute is substituted in a path expression. In this case, only a subset of objects in the codomain class of the attribute refer to the objects in the *substituted* domain. For example, suppose the domain of A_i is substituted by D_j in Fig. 2. Then, since C^* objects refer to the D^* objects, only a subset of C^* objects may refer to D_j objects. Thus, the codomain of A_i (e.g., C_i) partially participate in the join with D_j.

The second case occurs when the codomain of an attribute is substituted in a path expression. Then, only a subset of objects in the domain class of the attribute are referred to by the objects in the *substituted* codomain. For example, suppose the codomain of A_i is substituted by C_i, which is a subclass of C. Then, since D^* objects are referred to by C^* objects, only a subset of D^* objects are likely to be referred to by C_i objects. Thus, the domain of A_i (e.g., D_j) partially participates in the join with C_i. We justify this case more formally as follows. The distribution of the values of the attribute A_i across the classes in the class hierarchy rooted at C can be classified into three cases [14]. Here, we assume that the domain of the attribute A_i is D^*.

- disjoint: each D^* object is referred to by one class of C^*.

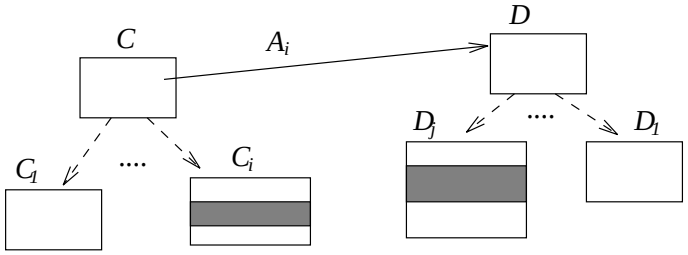

Fig. 2: A schema graph.

- total inclusive: each D^* object is referred to by any class in C^*.
- partial inclusive: the cases neither disjoint nor total inclusive.

Except for the total inclusive case, the second reason is satisfied because some objects of D^* are not referred to via $C_i.A_i$.

The third case occurs when a class in a path is the *common domain* of multiple attributes, each of which refers to only a subset of the objects in the domain class. For example, suppose that the class D_j in $C_i.A_i[D_j]$ is also the domain of the attribute $C'.A'_i$. Then, D_j is the common domain of the attributes $C_i.A_i$ and $C'.A'_i$. Here, since only a subset of D_j objects may be referred to by $C_i.A_i$ (the remainder of D_j objects being referred to by $C'.A'_i$), the domain of A_i (i.e., D_j) partially participates in the join with C_i.

For these reasons, object-oriented queries involve partial participation more frequently than relational ones. Let us note that, in Example 2, all the paths except for P_1, P_5, and P_9 include both codomain and domain substitutions, and thus involve partial participation join. Furthermore, in the path P_{12}: S.own[T].manuf.name, if few students have a small portion of trucks, as is likely to happen in a real world, then most of the objects in the Student and Truck become dangling objects. Thus, the effects of partial participation in object-oriented databases are more significant than in relational databases.

4. ESTIMATION OF INTERMEDIATE RESULTS CARDINALITIES

In this section, we clarify the effects caused by partial participation in detail and propose new techniques for estimating the cardinalities of unconditional joins, selectivity factors, and intermediate results cardinalities that take the effects into account. In Section 4.1, we discuss the problems of conventional techniques for estimating the cardinalities of unconditional joins when they are used for partial participation joins. In Section 4.2, we present a new estimation technique that takes the effect of partial participation and multi-valued attributes into account. In Section 4.3, we discuss the effect of partial participation on the estimation of selectivity factors for conditions and present a new selectivity estimation technique. In Section 4.4, we show how to estimate intermediate results cardinalities for the queries involving n-way joins with selection conditions by using the cardinalities of unconditional joins and the selectivity factors. We then compare our estimation techniques with other existing ones.

4.1. Problems of Conventional Techniques for Estimating the Cardinalities of Joins

Many techniques for estimating the cardinality of a join have been proposed in the literature for relational databases [18, 19, 21]. We use the relational join $C_i \bowtie_{A_i=A_j} D_j$ as an example. We assume that no selection conditions are applied to individual relations (classes) until Section 4.3.

Selinger et al. [19] assume (1) uniform distribution [4], (2) attribute independence [4], and (3) referential integrity; that is, each value in the join attribute with the smaller cardinality has a matching value in the other join attribute. Under these assumptions, they estimate the cardinality of the join as follows:

$$n(C_i \bowtie D_j) \quad = \quad n(C_i) \times n(D_j) \times \frac{1}{max[\, dv(C_i.A_i),\, dv(D_j.A_j)\,]} \tag{2}$$

Here, $n(C_i) \times n(D_j)$ is the cardinality of the Cartesian product between C_i and D_j, and $1/max[dv(C_i.A_i),\ dv(D_j.A_j)]$, the *selectivity factor* for the join, is the probability that a pair of tuples randomly selected from the Cartesian product satisfies the join condition. However, if the join is a partial participation join, the assumption (3) no longer holds. In Eq.(2), since $n(C_i) \geq dv(C_i.A_i)$ and $n(D_j) \geq dv(D_j.A_j)$, the estimated cardinality is always greater than or equal to $min[\, n(C_i),\ n(D_j)\,]$. Thus, the larger the portions of the dangling tuples in C_i and D_j become, the more error the estimation of the cardinality would produce. In an extreme, when only one pair of tuples from C_i and D_j participates in the join, the estimated cardinality is at least $min[\, n(C_i), n(D_j)\,]$ times larger than the actual count. Thus, a correction factor should be applied

to the estimates. We note that when assumption (3) as well as assumptions (1) and (2) holds, Eq. (2) gives the correct result.

Whang et al. [24, 21, 22] consider the effect of partial participation in estimating the cardinality of the join. They define the *join participation ratio* $J(C_i)$ (defined originally as the *join selectivity* [24]) for the relation C_i as the ratio of the number of distinct join column values of the tuples participating in the unconditional join to the total number of the distinct join column values of C_i. They also assume that the relationship between C_i and D_j is many-to-one.† Under this assumption, they estimate the cardinality for $C_i \bowtie_{A_i=A_j} D_j$ as follows:

$$n(C_i \bowtie D_j) \quad = \quad n(C_i) \times J(C_i) \times \frac{n(D_j)}{dv(D_j.A_j)} \tag{3}$$

Here, $n(C_i) \times J(C_i)$ is the number of tuples in C_i that participate in the join, and $\frac{n(D_j)}{dv(D_j.A_j)}$ is the average number of tuples in D_j that have the same value in the attribute A_j. In Eq. (3), since the join selectivity $J(C_i)$ decreases as the portion of dangling tuples in C_i increases, the effect from partial participation join is accounted for.

Since the estimation techniques proposed by Selinger et al. [19] and Whang et al. [24, 21, 22] are for the relational model, they have not considered the characteristics of object-oriented databases, such as multi-valued attributes, domain and codomain substitutions. Thus, these techniques are not appropriate for estimating the cardinalities of object-oriented queries.

In object-oriented databases, most literature assumes a simple query model in which no null values and no domain substitutions are allowed. Partial participation does not occur frequently in such a query model. Most of the research on query optimization and physical database design [2, 8, 11, 16] uses the average fan-out *fan* and the sharing level *share* as a part of the cost model. Let us consider a path expression $C_i.A_i[D^*]$ on the schema graph shown in Fig. 2. The fan-out, $fan(C_i.A_i)$, is the average number of D^* objects referred to by an object of C_i. Similarly, the sharing level, $share(C_i.A_i)$, is the average number of C_i objects that refer to the same object of D^*. By using these parameters, the cardinality of the join $C_i.A_i[D^*]$ can be estimated in two different ways depending on the directions of the traversal† as follows [2, 11]:

$$n(C_i \bowtie D^*) \quad = n(C_i) \times fan(C_i.A_i) \qquad /^* \text{ forward traversal } ^*/ \tag{4}$$

$$n(C_i \bowtie D^*) \quad = n(D^*) \times share(C_i.A_i) \quad /^* \text{ backward traversal } ^*/ \tag{5}$$

Bertino et al. [2] assume only single-valued attributes exist without null values and codomain and/or domain substitutions. Under these assumptions, $fan(C_i.A_i)$ is estimated as 1, and $share(C_i.A_i)$ as $\frac{n(C_i)}{dv(C_i.A_i)}$. Kim et al. [11] assume no null values and no domain substitutions exist. Under these assumptions, $fan(C_i.A_i)$ is estimated as 1 if A_i is a single-valued attribute; otherwise it is estimated as $\frac{n(C_i.A_i)}{n(C_i)}$. However, Kim et al. [11] have not proposed the sharing level. Kemper [8] and Lanzelotte [16] consider null values and multi-valued attributes with no codomain and/or domain substitutions. They estimate $fan(C_i.A_i)$ as 1 and $share(C_i.A_i)$ as $\frac{n(C_i)}{dv(C_i.A_i)}$ when A_i is a single-valued attribute. However, they assume that fan-outs and sharing levels for multi-valued attributes are supplied by users.

These conventional estimation techniques have the following problems if they are used for object-oriented queries involving partial participation joins or multi-valued attributes. First, when C_i is a partial participation class (i.e., $C_i.A_i$ has null values or the domain of $C_i.A_i$ is substituted), $fan(C_i.A_i)$ is likely to be less than 1 even though the attribute A_i is a single-valued attribute. For example, if only half of the C_i objects participate in the join, $fan(C_i.A_i)$ would be 0.5 instead of 1. Second, the parameter $\frac{n(C_i.A_i)}{n(C_i)}$ in Kim [11] has a fixed value independent of domain substitution of A_i, thus, cannot reflect the effect of domain substitution. A similar argument holds for the parameter $\frac{n(C_i)}{dv(C_i.A_i)}$ in Bertino [2], Lanzelotte [16], and Kemper [8]. Third, since most of the

† In the relational database design, a many-to-many relationship between entity sets is transformed into two one-to-many relationships between relations [6].

† There are two possible ways to traverse the classes in a path [11]: the *forward traversal* visits the classes from the codomain of an attribute to its domain, and the *reverse traversal* in the opposite direction.

conventional estimation techniques assume single-valued attributes, it is natural to expect errors in estimation when multi-valued attributes are involved in the query. For example, if A_i is a multi-valued attribute and has d values on the average, then $share(C_i.A_i)$ in Bertino [2], Lanzelotte [16], and Kemper [8], should be multiplied by d.

The following example shows the error in estimation when using conventional methods for object-oriented queries involving partial participation joins. (Selinger's and Whang's methods are not compared since they are developed for the relational model.)

Example 3 Consider the path Student.own[Truck] in Fig. 1. For convenience, we abbreviate the class names Student and Truck as S and R. We assume that the attribute own is single-valued and does not allow null values to conform to the assumptions of the conventional techniques [2, 8, 11, 16]. We also assume the following statistics:

$$\mathsf{n(S)} = 1{,}000;\ \mathsf{n(S.own)} = 1{,}000;\ \mathsf{dv(S.own)} = 500;\quad \mathsf{n(S.own[R])} = 50;\ \mathsf{n(R)} = 2{,}000.$$

Note that the join Student.own[Truck] is a partial participation join in which only 50 students (among 1,000 students) have less than 50 trucks (among 2,000 trucks) since n(S.own[R]) = 50, the attribute own is single-valued, and some trucks may be shared. Conventional techniques [2, 8, 16] estimate $fan(S.own)$ as 1 and $share(S.own)$ as $\frac{n(S)}{dv(S.own)} = \frac{1{,}000}{500}$ regardless of the actual number of trucks referenced by the attribute own. These estimates are correct only when all students refer only to trucks. The cardinality of the join using these parameters can be estimated as follows depending on the join directions [2, 11]:

$$\begin{aligned} n(S \bowtie R) &= n(S) \times fan(S.own) = 1{,}000 \times 1 = 1{,}000 && \text{/* forward traversal */} \\ n(S \bowtie R) &= n(R) \times share(S.own) = 2{,}000 \times \tfrac{1{,}000}{500} = 4{,}000 && \text{/* backward traversal */} \end{aligned}$$

However, since each of the 50 students have a truck, the correct value of n(S $\bowtie$ R) is 50.

4.2. Estimation of Cardinalities of Joins

In this section, we present a new technique for estimating the cardinalities of joins in object-oriented queries. This technique takes into account the effects of partial participation and multi-valued attributes.

Let us consider the join $C_i.A_i[D_j]$ shown in Fig. 2 again. For each codomain and domain pair (C_i, D_j) of the attribute A_i, we define the fan-out as the average number of D_j objects referred to by an object of C_i and denote it as $fan(C_i.A_i[D_j])$. Similarly, we define the sharing level as the average number of C_i objects that refer to the same object of D_j and denote it as $share(C_i.A_i[D_j])$. We calculate these parameters as follows:

$$fan(C_i.A_i[D_j]) = \frac{n(C_i.A_i[D_j])}{n(C_i)} \tag{6}$$

$$share(C_i.A_i[D_j]) = \frac{n(C_i.A_i[D_j])}{n(D_j)} \tag{7}$$

Here, since $n(C_i.A_i[D_j])$ is the total number of Oids (i.e., references) including duplicates in $C_i.A_i$ that refer to D_j objects, each C_i object has $\frac{n(C_i.A_i[D_j])}{n(C_i)}$ Oids of D_j objects on the average. Similarly, we calculate $share(C_i.A_i[D_j])$ as in Eq. (7). Fig. 3 shows $fan(C_i.A_i[D_j])$ and $share(C_i.A_i[D_j])$ graphically.

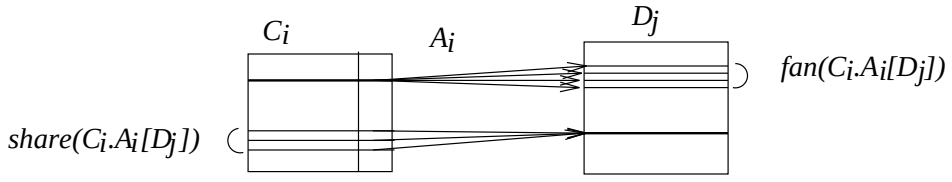

Fig. 3: $fan(C_i.A_i[D_j])$ and $share(C_i.A_i[D_j])$

The *fan* and *share* of Eqs. (6) and (7) consider partial participation and multi-valued attributes. For single-valued attributes, as the numbers of dangling objects in C_i and D_j increase, the *fan* and *share* defined in Eqs. (6) and (7) decrease proportionally. Specifically, if the attribute A_i has null values or the domain of A_i is substituted (i.e., C_i partially participates in the join), then $n(C_i.A_i[D_j])$ may become less than $n(C_i)$. Hence, $fan(C_i.A_i[D_j])$ may also become less than 1. Similarly, if the codomain of A_i is substituted or if the class D_j is a common domain (i.e., D_j partially participates in the join), then $share(C_i.A_i[D_j])$ may also become less than $\frac{n(C_i)}{dv(C_i.A_i)}$. For multi-valued attributes with d values on the average, the fan-outs and the sharing levels may be as much as d times greater than those of single-valued attributes. These results contrast with those of the previous methods, in which the fan-out and the sharing level are fixed as 1 and $\frac{n(C_i)}{dv(C_i.A_i)}$ (≥ 1) regardless of the presence of dangling objects.

By using the fan-out and the sharing level defined in Eqs. (6) and (7), we estimate the cardinality of the join $C_i.A_i[D_j]$ as follows. We use two formulas depending on the join directions:

$$n(C_i \bowtie D_j) = n(C_i) \times fan(C_i.A_i[D_j]) \quad /* \text{ forward traversal } */ \tag{8}$$

$$n(C_i \bowtie D_j) = n(D_j) \times share(C_i.A_i[D_j]) \quad /* \text{ backward traversal } */ \tag{9}$$

Example 4 We estimate the cardinality of the join in Example 3 using the proposed method. Note that these results match with the true cardinalities.

$$n(S \bowtie R) = n(S) \times fan(S.own[R]) = 1,000 \times \tfrac{50}{1,000} = 50 \quad /* \text{ forward traversal } */$$

$$n(S \bowtie R) = n(R) \times share(S.own[R]) = 2,000 \times \tfrac{50}{2,000} = 50 \quad /* \text{ backward traversal } */$$

Since the join is a partial participation join where students have only 50 trucks (including duplicates) among 1,000 vehicles (including cars, trucks, and duplicates), then the correct values of *fan* and *share* are $\frac{50}{1,000}$ and $\frac{50}{2,000}$, respectively; thus, the cardinality $n(S \bowtie R)$ becomes 50.

In Eqs. (6) and (7), the parameter $n(C_i.A_i[D_j])$ can be easily maintained in two ways: dynamically and statically. In dynamic maintenance, when inserting an object o into C_i, we increment $n(C_i.A_i[D_j])$ by k if the field $o.A_i$ contains k Oids of D_j. Selecting only Oid's of D_j is possible since each Oid includes the class identifier. When deleting an object o from C_i, we decrement $n(C_i.A_i[D_j])$ by k if $o.A_i$ contains k Oids of D_j. In the case of updates, the parameter can be maintained similarly: (1) If the old value in $o.A_i$ contains $k1$ Oids of D_j, we decrement $n(C_i.A_i[D_j])$ by $k1$; (2) If the new value in $o.A_i$ contains $k2$ Oids of D_j, we increment $n(C_i.A_i[D_j])$ by $k2$. In static maintenance, we count $n(C_i.A_i[D_j])$ by accessing the objects of C_i sequentially at a predefined interval or upon a user request. As each object is accessed, we add up the number of D_j's Oids contained in the attribute A_i. When this process finishes, the accumulated number of D_j's Oids becomes $n(C_i.A_i[D_j])$.

4.3. Selectivity Estimation for Conditions

Partial participation influences not only estimation of cardinalities for unconditional joins but also estimation of selectivities for conditions. In the previous section, we have not taken into account the selection conditions in the query imposed upon the classes to be joined. In this section, we discuss the problems of the conventional selectivity estimation techniques [19] when they are used for estimating the selectivities of conditions on partial participation classes. We then propose a new selectivity estimation technique that takes the effect of partial participation into account. The effect of multi-valued attributes is also taken into accounted in the proposed technique.

A *nested predicate* is a condition on a path expression of an arbitrary length [2]. If the length of the path expression is 1, we call it a *simple predicate* [2]. Since object-oriented queries include nested predicates, a query optimizer has to estimate the selectivities of the nested predicates called path selectivities. We define the *path selectivity* as the ratio of the number of path instances that satisfy the predicate to the total number of path instances, i.e., the probability that a randomly selected path instance satisfies the predicate.

As an example, let us consider the predicate $C_i.A_i[D_j].A_j =$ "*val*" in Fig. 4. Here, we assume that the domain of A_j is a primitive class. In Fig. 4, the grey-colored subregions (denoted by D'_j) of D_j denote the objects participating in the join with C_i, and the others non-participating objects. The selectivity of this nested predicate is the ratio of the number of path instances of the

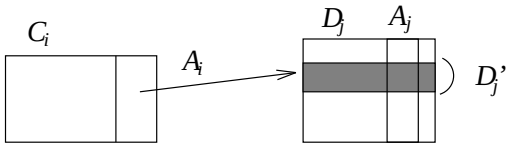

Fig. 4: Condition $C_i.A_i[D_j].A_j =$ "*val*".

path $C_i.A_i[D_j].A_j$ that satisfy $A_j =$ "*val*" to the total number of the path instances. Most of the conventional literature [2, 3, 9, 16, 17] on object-oriented databases have no explicit estimation techniques for path selectivity. Kim et al. [11] use the technique proposed by Selinger et al.[19] for estimating the path selectivity as follows:

$$sel(C_i.A_i[D_j].A_j = \text{"}val\text{"}) \quad = \quad sel(D_j.A_j = \text{"}val\text{"}) = \frac{1}{dv(D_j.A_j)} \tag{10}$$

However, if the codomain of the ending attribute (i.e., A_j) of the path expression (i.e., D_j) is a partial participation class, Eq. (10) produces an incorrect result. Specifically, since the path instances of the path expression $C_i.A_i[D_j].A_j$ contain only the values of A_j that belong to D'_j (denoted as $D'_j.A_j$), the probability that a path instance satisfies the predicate $C_i.A_i[D_j].A_j =$ "*val*" should be estimated over the objects in D'_j instead of those in the entire D_j. The following example illustrates this point.

Example 5 In Fig. 1, we assume that the number of companies that produce cars is *five* among 500 companies: dv(Car.manuf[Company]) = 5, n(Company) = 500. The selectivity for Car.manuf[Company].name = "Ford" is 1/5, since there are only *five* companies that produce cars. However, the conventional method using Eq. (10) estimates the selectivity as 1/500, which is equivalent to the selectivity of the simple predicate Company.name = "Ford".

We propose a new estimation technique for the path selectivity that considers the effects of partial participation. The key idea is to estimate the selectivity of the nested predicate $C_i.A_i[D_j].A_j =$ "*val*" within D'_j instead of the entire D_j as follows:

$$sel(C_i.A_i[D_j].A_j = \text{"}val\text{"}) \quad = \quad \frac{1}{dv(C_i.A_i[D_j].A_j)} = \frac{1}{dv(D'_j.A_j)} \tag{11}$$

Here, $dv(C_i.A_i[D_j].A_j)$ is the number of distinct values in $D_j.A_j$ that are referred to by $C_i.A_i$ (i.e., $dv(D'_j.A_j)$). Hence, $1/dv(C_i.A_i[D_j].A_j)$ is the probability that a path instance satisfies $A_j =$ "*val*". For example, in Example 5, Eq. (11) estimates the path selectivity for Car.manuf[Company].name = "Ford" as follows:

$$sel(Car.manuf.name = \text{"}Ford\text{"}) = \frac{1}{dv(Car.manuf.name)} = \frac{1}{5}.$$

As a special case, when the length of the path in a nested predicate is 1, Eq. (11) becomes the selectivity for the simple predicate, i.e., equivalent to Eq. (10).

Eq. (11) provides more accurate estimates than Eq. (10). For total participation (i.e., D_j in Fig. 4 totally participates), the results of Eq. (10) and Eq. (11) are identical since $dv(C_i.A_i[D_j].A_j)$ is equal to $dv(D_j.A_j)$. For partial participation, Eq. (11) estimates the selectivity greater than Eq. (10) by a factor of $\rho = \frac{dv(D_j.A_j)}{dv(C_i.A_i[D_j].A_j)}$ since $dv(D_j.A_j) \geq dv(C_i.A_i[\ D_j].A_j)$. As an example, in Example 5, since $\rho = \frac{dv(Company.name)}{dv(Car.manuf[Company].name)} = \frac{500}{5} = 100$, Eq. (11) estimates the selectivity *one hundred* times greater than Eq. (10).

Compared with conventional ones, the proposed method requires more delicate statistics. The statistics used in Eq.(11) is $dv(C_i.A_i[D_j].A_j)$; in contrast, the statistics used in Eq.(10) is $dv(D_j.A_j)$. The complexity of the former is higher than that of the latter. We propose two methods for computing these statistics efficiently: one is based on the nested index [2] and the other on the formula proposed by Yao [25].

(1) Method 1: Using a nested index

A nested index [2] establishes a direct connection between the first and the last objects in a path instance. It can be implemented using some variation of the B-tree data structure [2]. A non-leaf node consists of records, where each record is a triple (key length, key value, pointer). The pointer contains the physical address of the next-level index node. An index record in a leaf node consists of the record length, key length, key value, the list of Oids of the instances of the starting class that hold the key value in the indexed attribute (i.e., the ending attribute), and the number of elements in the list of Oids. A nested index *n-indx* defined on the path expression $C_i.A_i[D_j].A_j$ has $dv(C_i.A_i[D_j].A_j)$ key-values in the leaf nodes. Thus, with *n-indx* we can dynamically maintain the statistics as follows:

Algorithm 1 (Statistics Maintenance)

```
if a new key K1 is created in the n-indx
   increment dv(Ci.Ai[Dj].Aj) by one;
if an existing key K2 is deleted from the n-indx
   decrement dv(Ci.Ai[Dj].Aj) by one;
```

Other indices such as a *path index* [2] or an *access support relation* [8] can also be used instead of the nested index since these indices have more information than the nested index.

(2) Method 2: Using Yao's formula

If there is no nested indices on the path expression $C_i.A_i[D_j]A_j$, we can compute the statistics by using Eq. (12) originally proposed by Yao [25].

Theorem 1 [25]: *Given* n *records grouped into* m *blocks* ($1 \leq m \leq n$), *each containing* bf=n/m *records. If* k *records are randomly selected from the* n *records, the expected number of blocks hit (blocks with at least one record selected) is given by*

$$\begin{aligned} b(m, bf, k) &= m \times (1 - \frac{{}_{n-bf}C_k}{{}_nC_k}), \text{ when } k \leq n - bf, \\ b(m, bf, k) &= m, \text{ when } k > n - bf. \end{aligned} \tag{12}$$

Eq. (12) is originally intended to estimate the number of block accesses when a certain number of records are randomly selected. However, it can also be applied to estimating the number of distinct values in an attribute after selection operations [24]. We estimate $dv(C_i.A_i[D_j].A_j)$ by using Eq. (12) as follows:

$$dv(C_i.A_i[D_j].A_j) = b(dv(D_j.A_j), \frac{n(D_j.A_j)}{dv(D_j.A_j)}, dv(C_i.A_i[D_j])) \tag{13}$$

Eq. (13) can be explained as follows. Objects in class D_j are partitioned into $dv(D_j.A_j)$ groups, where each group has objects with the same value in the attribute A_j. These groups, called the *logical groups* [23], correspond to the number of blocks in Eq. (12). The parameter $\frac{n(D_j.A_j)}{dv(D_j.A_j)}$ represents the average number of objects in each group. It corresponds to the blocking factor in Eq. (12). The last parameter $dv(C_i.A_i[D_j])$ is the number of objects referred to by $C_i.A_j$ and corresponds to the number of records to be selected in Eq. (12).

Since Eq. (12) involves an iterative form, the computation cost can be excessively high as k becomes large. Approximation formulas for faster evaluation are presented in Whang et al. [23] and Diehr et al. [5].

4.4. Cardinalities of Intermediate Results for N-way Joins

In this section, we estimate intermediate results cardinalities for *n-way* joins (nested predicates involving n classes) by using the cardinalities of unconditional joins and selectivity factors presented in Sections 4.2 and 4.3, respectively.

A query involving n classes may be joined in any order among $n!$ permutations, just as n relations may be joined in any order of $n!$ permutations [12]. Let $< C_1, C_2, ..., C_n >$ be a join sequence (permutation) of the set of n classes in the query. We define the *k-th intermediate result*, denoted by T_k, in the join sequence $< C_1, C_2, ..., C_n >$ as the result of joining the partial join sequence $< C_1, C_2, ..., C_k >$. We also define the *k-th intermediate result cardinality*, denoted by $n(T_k)$, as the cardinality of T_k. We adopt the heuristic of avoiding Cartesian products as in [19]. Thus, among classes C_1,...,C_{k-1}, class $C_j, 1 \leq j \leq k-1$, is directly connected to the next class C_k [19]. Two classes are *directly connected* if one is the codomain and the other is the domain of an attribute. Then,

$$\begin{aligned} n(T_k) &= [\ n(T_{k-1})\ ] \times [\ \text{fan-out (or sharing level)}\ ] \times [\ \text{selectivity of } C_k\ ] \\ &= n(T_{k-1}) \times fan(C_j.A_j[C_k])\ \ (\text{or } share(C_k.A_k[C_j])) \times sel(C_k),\ \ k \geq 2 \\ n(T_1) &= n(C_1) \times sel(C_1) \end{aligned} \tag{14}$$

where $sel(C_k)$ is the product of selectivities of the selection conditions in the query that are applied to the class C_k.

Example 6 In Fig. 1, we estimate the intermediate results cardinalities for the nested predicate Student.own[Truck]manuf[Company].name = "Ford" as follows. For convenience, we abbreviate the class name Student, Truck, and Company as S, R, and C, respectively. We assume the following statistics given in Table 1.

Table 1: Statistics.

S (Student)	R (Truck)	C (Company)
$n(S) = 1,000$	$n(R) = 2,000$	$n(C) = 500$
$n(S.own) = 1,500$	$n(R.manuf) = 2,000$	$n(C.name) = 500$
$n(S.own[R]) = 50$	$n(R.manuf[C]) = 2,000$	$n(C.name[String]) = 500$
$dv(S.own[R]) = 50$	$dv(R.manuf[C]) = 5$	$dv(C.name[String]) = 500$

1) $dv(S.own[R].manuf[C].name)$

- $dv(S.own[R].manuf[C])$
  $= b(dv(R.manuf[C]), \frac{n(R.manuf[C])}{dv(R.manuf[C])}, dv(S.own[R])) = b(5, 400, 50) \simeq 5$
- $dv(S.own[R].manuf[C].name)$
  $= b(dv(C.name), \frac{n(C.name)}{dv(C.name)}, dv(S.own[R].manuf[C])) = b(500, 1, 5) \simeq 5$

2) Selectivities

- $sel(S) = 1$ • $sel(R) = 1$
- $sel(C) = sel(S.own[R].manuf[C].name = \text{"}Ford\text{"}) = \frac{1}{dv(S.own[R].manuf[C].name)} = \frac{1}{5}$

3) Intermediate results cardinalities in the join sequence of (S, R, C)

- $n(T_1) = n(S) \times sel(S) = 1,000 \times 1 = 1,000$
- $n(T_2) = n(T_1) \times fan(S.own[R]) \times sel(R) = 1,000 \times \frac{n(S.own[R])}{n(S)} \times 1 = 50$
- $n(T_3) = n(T_2) \times fan(R.manuf[C]) \times sel(C) = 50 \times \frac{n(R.manuf[C])}{n(R)} \times \frac{1}{5} = 10$

Actually, since there are 1,000 students and $sel(S) = 1$, $n(T_1) = 1,000$. Since 1,000 students own 50 trucks ($n(S.own[R]) = 50$) and $sel(R) = 1$, $n(T_2) = 50$. Since each truck has a manufacturer

and only *one fifth* of them satisfy the condition "= Ford", $n(T_3) = 10$. These are equivalent to our results.

In contrast, if we use conventional methods, the following are the cardinalities of intermediate results produced by using Eqs. (4) and (5). Note that conventional methods estimate $sel(C)$ as $1/500$.

- $n(T_1) = n(S) \times sel(S) = 1,000 \times 1 = 1,000$
- $n(T_2) = n(T_1) \times fan(S.own) \times sel(R) = 1,000 \times \frac{n(S.own)}{n(S)} \times 1 = 1,500$
- $n(T_3) = n(T_2) \times fan(R.manuf) \times sel(C) = 1,500 \times \frac{n(R.manuf)}{n(R)} \times \frac{1}{500} = 3$

These results show significent deviations from the true values for $n(T_2)$ and $n(T_3)$.

Based on these intermediate result cardinalities, we are able to estimate the cost of an evaluation plan for the nested predicate Student.own[Truck]manuf[Company].name = "Ford". For simplicity, we consider the evaluation plan of forward traversal pointer-based nested-loop join [11, 20]. Table 2 shows the comparison of the cost estimations. Here, the cost model proposed by Kim et al. [11] is used. Other join algorithms such as the sort-domain join or the hash join [11, 20] also

Table 2: Cost estimation for an evaluation plan.

	proposed method	conventional methods
$cost(T_1)$	page(S) : number of page accesses for Student	page(S)
$cost(T_2)$	$n(T_1) \times fan(S.own[R])$ $= 1,000 \times \frac{50}{1,000}$ $= 50$ page accesses	$n(T_1) \times fan(S.own)$ $= 1,000 \times \frac{1,500}{1,000}$ $= 1,500$ page accesses
$cost(T_3)$	$n(T_2) \times fan(R.manuf[C])$ $= 50 \times \frac{2,000}{2,000}$ $= 50$ page accesses	$n(T_2) \times fan(R.manuf)$ $= 1,500 \times \frac{2,000}{2,000}$ $= 1,500$ page accesses
evaluation cost	page(S)+50+50 page accesses	page(S)+1,500+1,500 page accesses

result in similar outcomes. The results show significant difference in cost estimation, which could lead the query optimizer to choose a solution that is far from the optimal one.

5. CONCLUSIONS

We have presented a new technique for estimating the intermediate results cardinalities of a query involving partial participation and multi-valued attributes. Although query optimization has been studied extensively in the literature, partial participation has not been considered seriously. The contributions of the paper can be summarized as follows.

First, we have argued that an object-oriented query often involves partial participation of classes in a relationship and compare the case with that for relational queries. Main causes of partial participation are codomain/domain substitutions and common domains, which are the characteristics of object-oriented databases.

Second, we have shown that conventional techniques may involve large estimation errors in estimating the intermediate results cardinalities when the query includes partial participation classes or multi-valued attributes. We have proposed new techniques for estimating the intermediate results cardinalities in which the effects of partial participation and multi-valued attributes are taken into accounted.

Finally, we have presented efficient methods for obtaining detailed statistics to be used for accommodating partial participation. Although these statistics are difficult to obtain in the relational databases, the proposed methods efficiently obtain those statistics by using inherent properties – such as object identifiers and nested indices – of object-oriented databases.

The most significant advantage of our technique is that the accuracy of the estimate for the cardinalities of unconditional joins and the selectivity factors is far enhanced with only a small overhead. Thus, the optimizer will be able to estimate the cost of an execution plan more accurately. As further studies, we are extending the selectivity estimation technique to consider range queries and are developing cost models for various other join algorithms.

Acknowledgements — We thank the reviewers for their careful comments on an earlier manuscript that helped enhance the clarity of this paper significantly. This work was partially supported by the Korean Ministry of Information and Communications through Korea Computer and Communications Co. as a part of the ODYSSEUS OODBMS project. It was also partially supported by the Korea Science and Engineering Foundation (KOSEF) through the Center for Artificial Intelligence Research. The last author was supported by the Brain Pool program of the Korean Foundation of Science and Technology Societies (KOFST). We would like to acknowledge the careful comments provided by Dr. K. H. Hong.

REFERENCES

[1] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system menifesto. In *Proc. Int. Conf. on Deductive and Object-Oriented Databases*, pp. 40–57, Kyoto, Japan (1989).

[2] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Trans. on Knowledge and Data Engineering*, **1**(2):196–214 (1989).

[3] J. A. Blakeley, W. J. McKenna, and G. G. Graefe. Experiences building the Open OODB query optimizer. In *Proc. ACM SIGMOD Conference*, pp. 287–296, Washington, DC (1993).

[4] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Trans. on Database Systems*, **9**(2):163–186 (1984).

[5] G. Diehr and A. N. Saharia. Estimating block accesses in database organizations. *IEEE Trans. on Knowledge and Data Engineering*, **6**(3):497–499 (1994).

[6] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 2nd edition (1994).

[7] Y. S. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proc. ACM SIGMOD Conference*, pp. 268–277 (1991).

[8] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. ACM SIGMOD Conference*, pp. 364–374, Atlantic City, New Jersey (1990).

[9] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. In *Proc. ACM SIGMOD Conference*, pp. 336–347, Minneapolis, Minnesota (1994).

[10] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. ACM SIGMOD Conference*, pp. 393–402, San Diego, CA. (1992).

[11] K. C. Kim, W. Kim, D. Woelk, and A. Dale. Acyclic query processing in object-oriented databases. In *Proc. Intl. Conf. on Entity-Relationship Approach* (1988).

[12] W. Kim. *Introduction to Object-Oriented Databases*. MIT Press (1990).

[13] W. Kim. Object-oriented database systems: Promises, reality, and futures. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, pp. 255–280. Addision-Wesley (1995).

[14] W. Kim, K. C. Kim, and A. Dale. Indexing techniques for object-oriented databases. In W. Kim and F. Lochovsky, editors, *Object-oriented Concepts, Applications, and Databases*, pp. 371–394. Addision-Wesley (1989).

[15] K. H. Korth and A. Silberschatz. *Database System Concepts*. McGraw-Hill (1991).

[16] R. S. G. Lanzelotte and J-P. Cheiney. Adapting relational optimization technology to deductive and object-oriented declarative database languages. In *Proc. Intl. Workshop on Database Programming Languages*, pp. 322–335 (1991).

[17] D. Maier, S. Daniels, T. Keller, B. Vance, G. Graefe, and W. McKenna. Challenges for query processing in object-oriented databases. In J. C. Freytag, D. Maier, and G. Vossen, editors, *Query Processing for Advanced Database Systems*. Morgan Kaufmann (1994).

[18] M. V. Mannino, P. Chu, and T. Sagar. Statistical profile estimation in database systems. *ACM Computing Surveys*, **20**(3):191–221 (1988).

[19] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. ACM SIGMOD Conference*, pp. 23–34, Boston, Mass. (1979).

[20] E. Shekita and M. Carey. A performance evaluation of pointer-based joins. In *Proc. ACM SIGMOD Conference*, pp. 364–374, Atlantic City, New Jersey (1990).

[21] K. Y. Whang. Constructing cost formulas for relational database query optimizers: A tutorial. In *IEEE TENCON*, pp. 132–141, Seoul, Korea (1987).

[22] K. Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear time probabilistic counting algorithm for database applications. *ACM Trans. on Database Systems*, **15**(2):208–229 (1990).

[23] K. Y. Whang, G. Wiederhold, and D. Sagalowicz. Estimating block accesses in database organizations – a closed noniterative formula. *Comm. of the ACM*, **26**(11):940–944 (1983).

[24] K. Y. Whang, G. Wiederhold, and D. Sagalowicz. Separability – an approach to physical database design. *IEEE Trans. on Computers*, **33**(3):209–222 (1984).

[25] S. B. Yao. Approximating block accesses in database organizations. *Comm. of the ACM*, **20**(4):260–261 (1977).