An Object-Oriented Database System Jasmine: Implementation, Application, and Extension

Hiroshi Ishikawa, Member, IEEE, Yasuo Yamane, Yoshio Izumida, Member, IEEE, and Nobuaki Kawato, Member, IEEE

Abstract—New applications such as engineering tasks require complex object modeling, integration of database and programming facilities, and extensibility. We have devised an Object-Oriented DBMS called Jasmine for such advanced applications. This paper describes the implementation, application, and extension of Jasmine in detail. First, we focus on the impact of the design of its Object-Oriented model and language on database implementation technology. We describe what part of traditional relational database technology we extend to handle Object-Oriented features such as object identifiers, complex objects, class hierarchies, and methods. We introduce nested relations to efficiently store and access clustered complex objects. We use hash-based methods to efficiently access nonclustered complex objects. We provide user-defined functions directly evaluated on page buffers to efficiently process method invocation. We devise Object-Oriented optimization of queries including class hierarchies, complex objects, and method invocation. We incorporate dedicated object buffering to allow efficient access to objects through object identifiers. Second, we describe nontrivial applications of Jasmine and discuss the validity of Object-Oriented databases. We focus on a constraint management facility, which can be implemented by taking advantage of the extensibility of Jasmine. The facility includes constraint rules, called design goals, for automatic database population required by engineering applications. Third, we describe a view facility for schema integration also needed by engineering applications in distributed environments. We focus on how we extend Jasmine to implement the facility.

Index Terms—Advanced applications, constraint management, data model, database programming language, nested relations, object buffering, Object-Oriented databases, query language, query optimization, view mechanism.

1 INTRODUCTION

COMPLEX, large-scale database applications such as CAD [18], hypermedia [17], and AI [14], [15] have emerged. To accommodate these, we need next-generation of databases as a software platform, which can provide advanced functions such as complex object modeling and programming. Instead, Object-Oriented databases (OODBs) [9], [23], [31] are expected to be the next-generation of databases. In this paper we describe how we designed and implemented an Object-Oriented DBMS called Jasmine [16], [18], [19], [21]. We also discuss how we applied Jasmine to engineering tasks to verify its validity and how we extended Jasmine for such advanced applications.

In our previous paper [21], we described the overall functionality and implementation of Jasmine. This paper has different contributions as follows:

- First, we focus on the impact of the design of its Object-Oriented model and language on database implementation technology. We describe what part of traditional relational database technology we extend to handle Object-Oriented features such as object identifiers, complex objects, class hierarchies, and methods. We introduce nested relations to efficiently store and access clustered complex objects. We use hash-based methods to efficiently access nonclustered complex objects. We provide user-defined functions directly evaluated on page buffers to efficiently process method invocation. We devise Object-Oriented optimization of queries including class hierarchies, complex objects, and method invocation. We incorporate dedicated object buffering to allow efficient access to objects through object identifiers.

- Second, we describe nontrivial applications of Jasmine in detail and discuss the validity of Object-Oriented databases. We focus on a constraint management facility, which can be implemented by taking advantage of the extensibility of Jasmine. The facility includes constraint rules, called design goals, for automatic database population required by engineering applications [22].

- Third, we describe a view facility for schema integration needed by engineering applications in distribu-
uted environments. We focus on how we extend Jasmine to implement the facility.

This paper is organized as follows. Section 2 describes the object model and the object manipulation language of Jasmine. Section 3 describes the implementation of Jasmine. Section 4 discusses an Object-Oriented database approach to engineering applications. Section 5 describes schema translation by view as an extension to Jasmine. Section 6 compares our system with other related work and gives concluding remarks.

2 FUNCTIONALITY

2.1 Data Model

In this section, we will briefly describe Jasmine's object model (see [21] for the formal semantics). Objects are a collection of attributes, which are categorized into properties (enumerated attributes) and methods (procedural attributes). Properties are object structures and methods are operations on objects. Objects are categorized into instances and classes. Instances denote individual data and classes denote types (i.e., structures) and operations applicable to instances of the class. Instances consist of a collection of attribute names and values. Classes consist of attribute names, definitions, and associated information such as demons. Objects are uniquely identified by values of the system-defined attribute object identifier (OID). On the other hand, values such as numbers and character strings have no OIDs but do have classes defined by the system. Objects with OIDs are called reference objects and objects with no OIDs are called immediate objects. Objects can include other objects (i.e., OIDs) as attribute values. This enables the user to directly define complex objects (composite objects) [23], which supports aggregation directly.

Classes are organized into a hierarchy (more strictly, a lattice) by generalization relationships. This hierarchy is called a class hierarchy. Classes (i.e., subclasses) can inherit attribute definitions from their superclasses. The user can make instances (i.e., instantiate) from any class in a class hierarchy unlike Smalltalk-80 [11]. Such instances are called intrinsic instances of the class. Classes not only define object types and methods, but they are also interpreted as a set of instances, which supports classification. That is, the instances of a class is the union of all the intrinsic instances of itself and all its subclasses. This differentiates Jasmine from other OODBs such as GemStone [31] where the user must define separate classes both as type and as a set. Objects can have a set of objects as well as a singleton object as an attribute value. The former are called multiple-valued attributes and the latter single-valued attributes.

Specialized functions called demons can be attached to attributes. Constraint demons are checked before values are inserted into attributes. The values are only inserted if the demon returns true. If needed, if-added, if-removed, and if-updated demons are invoked when values are referenced, inserted, deleted, and replaced. Before and after demons are invoked before and after the procedural attributes they are attached to are invoked. The user can combine these demons to flexibly implement active databases [22], [35]. Unlike other systems, Jasmine allows the user to specify both system-defined integrity constraints such as mandatory, multiple, and to specify user-defined integrity constraints as demons.

Consider the class PATIENT as an example (see Fig. 1). The keyword Enumerated is followed by the definition of user-supplied enumerated attributes. The name facet such as Doctor denotes the name of an attribute. The class facet before the attribute name denotes the range class such as FLOA of Height. The value of the attribute of an instance must be an instance of the range class (see Fig. 2). The domain of the attribute is the class being defined, PATIENT. The multiple facet denotes that the attribute is multiple-valued such as Temperature. The mandatory facet denotes that the attribute allows no null value such as Doctor and Weight. The mandatory attribute must be specified its value at instantiation. The common facet denotes that the attribute value is common to all the instances as the domain objects. The common attribute is not necessarily a constant, such as Cardinality. The default facet contains a default value referenced when the attribute value is not yet specified, such as Category of PATIENT. The if-needed demon, invoked if the referenced attribute has a null value, computes a value such as Height of PATIENT. The keyword Procedural is followed by the definition of procedural attributes. Procedural attributes such as make-medical-certificate also have facets. The class facet such as MEDICALCERTIFICATE denotes the range class of the procedural result.

A superclass in a class hierarchy is denoted by the system-defined attribute Super. The superclass, for example, PERSON, includes its subclasses, PATIENT, as a set. The attributes of the superclass are inherited to the subclass, such as Age of PATIENT. An attribute can be newly defined in the subclass such as Doctor of PATIENT. Intrinsic instances of a nonleaf class can represent incomplete knowledge of the domain. For example, PERSON intrinsic in-
stances directly denote a set of persons known to be neither a patient nor a doctor.

A class can be divided into disjoint subclasses (see Fig. 3). Those subclasses are collectively called a partition. Each subclass is called a member of the partition. For example, PERSON has a partition consisting of DOCTOR and PATIENT. A partition denotes a categorization based on one viewpoint. Different viewpoints generate different partitions. PERSON has another partition of ADULT and CHILD. Members of distinctive partitions may not be disjoint, such as PATIENT and ADULT. Categorization conditions can be explicitly specified to make the partition semantics clear such as “Age < 18” of CHILD. Then the attribute Age is called a categorization attribute. The categorization conditions are basically specified by a subset of the database programming language described in Section 2.2. They can be used by query optimization described in Section 3.

Fig. 3. Example of a class lattice.

Jasmine allows a class to have multiple superclasses. They must be non-disjoint members of different partitions of a class. For example, ADULT-PATIENT inherits Doctor from PATIENT and Occupation from ADULT (multiple-inheritance [37]). Multiple superclasses sometimes have properties of the same name. We must resolve such conflicts. Assuming that the class of a property of a subclass should be either the class of or its subclass of the same property of a superclass, the rules are as follows:

1) A property with the most specific class is chosen if there is only one such property.
2) Otherwise, a property with the most specific class first found by a system-defined search (i.e., depth-first search) is chosen.
3) A property other than the one determined by the rule 1 or rule 2 can be chosen by explicitly specifying the superclass in the property definition if necessary.

2.2 Database Programming Language

This section describes an object manipulation language called Jasmine/C [21] as a database programming language which integrates a general-purpose programming language (C) and a database language in an Object-Oriented context, and which allows the user to program advanced applications. In Jasmine, the user manipulates objects by sending messages to objects just as in Object-Oriented programming languages. This type of access is called singleton access or individual access. The user can assign values to attributes and reference attribute values, Jasmine allows set-oriented access in addition to singleton access. Set-oriented access is done by a query on objects. The query language of Jasmine has the following features different from those of SQL [6]. The semantics can be formally defined through query translation by object operators as described in [21].

The basic unit of a query expression consisting of target and condition parts is an object expression, a class name followed by a series of attribute names. The target part is an object expression, or a list of object expressions. The condition part consists of a logical combination of predicates which compare object expressions. A query on a class returns all the instances of the class and its subclasses, so the user can retrieve by a single Jasmine query what would take multiple relational database queries to retrieve. The object expressions denote object joins. The object expressions can also contain methods, so the user can manipulate objects set-theoretically and filter a set of objects procedurally. If a superclass is specified with a method in a query, methods dedicated to instances of the class and its subclasses can be invoked simultaneously. This facilitates polymorphism [37] in a set-oriented manner. The system-defined methods such as put, delete specified in a query can modify a set of objects. A query can invoke demons which implement integrity facilities introduced by QBE [41]. The user can specify multiple-valued attributes in a query. The user can control unnesting of multiple values and apply aggregate functions correctly. Multiple-valued attributes are existentially or universally quantified.

The integration of query and programming facilities is another important feature for advanced applications. First, the user can specify methods in a query as described above. The user can extend the functionality of the query language just by defining and specifying a method in a query, without modifying the query language processor. The user can develop application programs more compactly without specifying details such as iteration variable declaration and control structures. Making this type of iteration implicit can increase physical data independence [6] of application programs by allowing the system to optimize the query expression. Second, the user can also define methods by specifying a query for them. This can define so-called virtual attributes and increase logical data independence [6] of application programs when applications evolve. Third, the fact that the user invokes a query from programs is one of salient aspects of advanced applications. We introduce set variables to solve the impedance mismatch problem [6] between the query and programming languages. The set variable has a class defined by an object model as its type and can contain a set of objects returned by a query as its value. The user can fetch an object by sending the scan message to the set variable and operate on the object by sending a message to the object in an Object-Oriented programming manner.

Class objects can also be operated set-theoretically for advanced applications. Basic database functions such as transactions, locking, and logging can be provided through system-defined classes. Multimedia data types and operations are provided by implementing them from system-defined primitive classes in a bootstrap manner.

Now we describe the syntax and semantics of a query through examples. The query has the following syntax:

```
"["object_expression(s)"]"
[where condition]
[groupby object_expression(s)]
```
where the object expression has the form:

\[
\text{class\_name} [\ldots .\text{"attribute\_name"} [\ldots .\text{"attribute\_name"} \ldots ]]
\]

The query expression evaluates to a set of the target objects satisfying the condition. The elements of the constructed set are objects (OIDs), or values belonging to the database, or newly constructed tuple values. The result type is determined by the query. For example, to find the name and address of inpatients, the user forms a query as follows:

**QUERY 1. [PATIENT.Name, PATIENT.Address] where PATIENT.Category == "inpatient"**

The tuple operator \( [ \ldots ] \) allows the construction of tuple values, corresponding either to projection or to join of relations. Like this example, the query corresponds to projection only if the target list has the following form:

\[
\{ \text{common\_object\_expression} .\text{attribute1}, \text{common\_object\_expression} .\text{attribute2}, \ldots \}
\]

Immediate objects are compared by ==, !=, >, >=, <, and <=, based on values.

In general, joins are categorized into implicit and explicit joins. Jasmine supports implicit joins as follows:

**QUERY 2. PATIENT.Doctor.Name where PATIENT.Name == "James Bond"**

This finds the name of doctors who are in charge of James Bond. The operator \( [ \ldots ] \) can be omitted only if the target list contains only one object expression. Assuming that \( C \) is a class and \( Ai \) is an attribute and \( Oi \) is an object, an implicit join denoted by an object expression \( C .Ai \). ... \( Ai \) has the following semantics:

\[
\{ Oi \mid Oi \text{ belongs to } C \text{ and for all } i = 1, \ldots, n, \text{ either of the following holds:}
\]

1) \( Oi \) is equal to \( Ai \) of \( Oi-1 \) if \( Ai \) is single-valued
2) \( Oi \) belongs to \( Ai \) of \( Oi-1 \) if \( Ai \) is multiple-valued

Nested sets generated by multiple-valued attributes are automatically flattened unless the user prohibits that.

Jasmine can also support explicit joins as follows:

**QUERY 3. [PATIENT.Name, DOCTOR.Name] where PATIENT.Address == DOCTOR.Address**

This retrieves pairs of names of patients and doctors who happen to live in the same area. \( [ \ldots ] \) in this case corresponds to join. Reference objects can also be compared by == and != based on OIDs. For example, assume Disease and Specialty are reference attributes (see Fig. 4):

**QUERY 4. [PATIENT.Name, DOCTOR.Name] where PATIENT.Disease == DOCTOR.Specialty**

This query finds the names of patients and doctors who specialize in their disease.

The object expression with multiple-valued attributes evaluates to a set of sets. However, multiple-valued attributes are automatically unnestesd unless the user specifies the prohibition of unnesting by a special operator described later. Therefore, the following query retrieves a flattened set of temperatures of serious patients:

**QUERY 5. PATIENT.Temperature where PATIENT.Condition == "serious"**

A condition on multiple-valued attributes is interpreted as at least one value satisfying the condition, that is, existentially. Universally-quantified multiple attributes can also be specified as described later. The following retrieves the names of patients who ran a temperature of higher than 38.0 degrees centigrade at least once:

**QUERY 6. PATIENT.Name where PATIENT.Temperature > 38.0**

Any class, leaf or nonleaf, in a generalization lattice can be specified in a set-oriented query. According to the interpretation of a class, the intrinsic instances of a nonleaf class and the instances of its subclasses can be retrieved at the same time. For example, to find persons who live in Tokyo:

**QUERY 7. PERSON where PERSON.Address == "Tokyo"**

This causes a query be specified compactly because several queries against subclasses such as PATIENT and DOCTOR can be formulated in a single query.

Objects can be retrieved without precise specification since a general class can be specified in a query together with an attribute defined in its subclasses. In general, assuming that \( C \) and \( C' \) are classes and \( A \) is an attribute, \( C \) is systematically translated into \( C' \) in a query only if the following set is not empty: \( \{ C' \mid C' \text{ is a subclass of } C \text{ and } A \text{ is defined or inherited by } C' \} \). The original query usually generates multiple queries. Note that Query 7 is a special case where \( A \) (e.g., Address) is defined or inherited by all classes in a class hierarchy with \( C \) (e.g., PERSON) as its top. For example, to find the names of persons whose disease is a fracture:

**QUERY 8. PERSON.Name where PERSON.Disease.Name == "fracture"**

The class PERSON is automatically specialized to the subclass PATIENT with the attribute Disease defined. In the extreme case, OBJECT can be used in a query. This mechanism fits with how we define some concepts by differentiating a general concept by providing specializing attributes. The user can thus formulate a query without knowing specificity like a natural language query.

A condition can be imposed on the categorization attribute of a general class with a partition. If the specified condition matches some of the categorization conditions of the partition, the specified class can be specialized to some of the partition members. In general, assuming that \( C \) and \( C' \) are classes, \( C \) is systematically translated into \( C' \) only if the following set is not empty: \( \{ C' \mid C' \text{ is a subclass of } C \text{ and the condition of a query and the categorization condition of } C' \text{ are not exclusive} \} \). For example, to find infants (i.e., younger than seven):

![Fig. 4. Part of a medical database structure.](image-url)
QUERY 9. PERSON where PERSON.Age < 7
The class PERSON is automatically specialized to CHILD with its categorization condition Age < 18.

The user can do operations other than retrieval set-theoretically by using procedural attributes, which can be specified in any part of an object expression of a query. The additional parameters of the procedural attribute are given in parentheses. In general, the object expression has the following form: receiver.method (parameter, parameter, ...). A receiver is an object expression and a parameter is an object expression, an object, or a value. The result is also an object, a value or a set of objects or values. For example, to make and print a copy of serious patients' medical certificates dated August 26, 1992, the user formulates the following query:

QUERY 10.
PATIENT.make-medical-certificate(“19920826”).print()
where PATIENT.Condition == “serious”

If we operate on objects set-theoretically in a setting other than Jasmine, we then have to retrieve a set of objects and scan and operate on each object in an iteration construct. In contrast, Jasmine makes this type of iteration implicit and iteration variable declaration unnecessary and allows the user to compactly specify a query without knowing the details.

We can also specify procedural attributes in incomplete knowledge access. If we specify a general class whose subclasses have procedural attributes of the same interface which have different implementations, the different attributes are invoked in a single query at the same time. In general, assuming that C and C’ are classes and M is a method, C is systematically translated into C’ only if the following set is not empty: \{C’ \in C’ | C’ is a subclass of C and M is defined or inherited by C’\}. M has different implementation depending on C’, so this realizes polymorphism in a set-oriented manner. For example, to display all heterogeneous media objects belonging to James Bond, such as X-ray and CT, the user specifies the following query:

QUERY 11. MEDIA.display() where MEDIA.Patient.Name == "James Bond"

Both system-defined and user-defined procedural attributes can be specified in the same way unlike other systems such as [10]. The system-defined procedural attributes include print and object modification operations such as put, replace, delete, and destroy. Of course, they can be invoked in a set-oriented query. In other words, the user can extend the query language without changing the parser or the code generator. For example, the following query adds 37.5 degrees centigrade to James’ temperature (multiple-valued attribute):

QUERY 12. PATIENT.put(“Temperature”, 37.5) where PATIENT.Name == "James Bond"

Attributes taking a set of objects and giving a singleton, called aggregate functions, can be specified in a set-oriented query. They include the system-defined attributes such as count, average, sum, max, and min. Since a set in our context allows duplication of objects, the user can use the aggregate functions naturally. For example, to find the number of inpatients who are under 7 years of age, the user forms a query as follows:

QUERY 13. PATIENT.count() where PATIENT.Age < 7 and PATIENT.Category == “inpatient”

The following query finds the average of the ages of inpatients,

QUERY 14. PATIENT.Age.average() where PATIENT.Category == “inpatient”

In general, the aggregate functions take as input the whole flattened set retrieved just before the function evaluation. This can cause subtle problems when the user applies the aggregate functions to the multiple-valued attribute. Assuming that the attribute Temperature is multi-valued, consider the following:

QUERY 15. PATIENT.Temperature.average() where PATIENT.Category == “inpatient”

This evaluates to the average of an automatically normalized set of objects as the values of Temperature of more than one inpatient. Therefore, if the user wants to apply the average to Temperature values of each PATIENT object, the user specifies a special operator \{[\] \} to prohibit automatic unnesting of multiple-valued attributes as follows:

QUERY 16. PATIENT.[Temperature].average() where PATIENT.Category == “inpatient”

Multiple-valued attributes can be universally quantified by specifying “All” before comparison operators. The following query retrieves the names of patients whose temperatures are all over 38.0 in contrast to Query 6:

QUERY 17. PATIENT.Name where PATIENT.[Temperature].All > 38.0

Grouping is allowed. The following calculates average temperatures for each group of inpatients of the same age:

QUERY 18. PATIENT.Temperature.Average() where PATIENT.Category == “inpatient” groupby PATIENT.Age

Procedural attributes can also be specified in object expressions in the condition part of a query to filter objects procedurally. This is powerful in a variety of applications. A content-based search of multimedia data can be done by defining an attribute such as like. The following finds a patient whose CT looks like sample-1 containing some disease:

QUERY 19. CT.Patient where CT.like(sample-1) == true

Usually the syntactically same object expressions in a query have the same semantics. However, making aliases of object expressions is possible anywhere in a query by qualifying them if necessary, for example, to do self-join. The following query retrieves pairs of patients who suffer from the same disease. Note that the second condition eliminates duplication.

QUERY 20. [P1: PATIENT, P2: PATIENT] where P1.PATIENT.Disease == P2.PATIENT.Disease and P1.PATIENT.Id < P2.PATIENT.Id

Now we can define the semantics of a query by using the semantics of object expressions defined earlier. First, the condition part is evaluated as follows: If simple conditions comparing object expressions are evaluated to be true,
based on the values of the object expressions, the Boolean combination of them is evaluated in the usual way. If the result is true, we then evaluate the target part and we get a tuple of objects or values as a query result.

It is necessary to individually access a set of retrieved objects in application programs. To this end, we introduce a variable which can be bound to a set of objects. The variable is called a set variable. The set variable is usually set to the result of a set-oriented query. The user can access objects one by one by scanning the set variable. The instance variable is also introduced to hold a singleton. The instance variable holds a single object like usual variables in a conventional programming language. The instance variable and set variable constitute the object variable. The object variable integrates set-oriented access of a database system and singleton access of a programming language. The existence of a multiple option at declaration specifies that the object variable is a set variable. For example,

\[
\text{PATIENT ps multiple, p;}
\]

ps and p are declared as set variable and instance variable of PATIENT type. In general, the set variable is set to the result set of a set-oriented query at the right-hand side of a statement. The user can access objects individually by using the system-defined procedural attributes as follows:

\[
\text{ps = PATIENT where PATIENT.Age < 13;}
\]

\[
\text{ps.openscan();}
\]

\[
\text{while (p = ps.next())}
\]

\[
\text{...}
\]

\[
\text{ps.closescan();}
\]

The procedural attribute next returns an object at each invocation, which is set to the instance variable p for further use.

Procedural attributes can include set-oriented queries. The following attribute of the class DEPARTMENT defines interns who work in a department:

Procedural DOCTOR intern() multiple
  \{ self.Doctor where self.Doctor.Status ==
  \"internship\"
\}

This can be specified in a query to retrieve interns in the pediatrics department as follows:

\[
\text{DEPARTMENT.intern() where}
\]

\[
\text{DEPARTMENT.Name == \"pediatrics\"}
\]

We don’t provide a special syntax for nesting queries. Instead, nested queries can be simulated by procedural attributes defined by queries like the above example. Correlated queries can be formulated explicitly by passing object expressions as parameters to the procedural attributes or implicitly through the system-defined variable self.

3 IMPLEMENTATION

Relational databases have already accumulated large amounts of implementation technology. We don’t think that it is clever to throw it away and to build Object-Oriented databases from scratch. Relational technology provides basically applicable techniques such as storage structures, access methods, query optimization, transaction and buffer management, and concurrency control. Therefore, we take a layered architecture consisting of object management and data management and use relational technology as data management (see Fig. 5). However, traditional relational technology has limitations in efficient support for Object-Oriented concepts including object identifiers, complex objects, class hierarchies, and methods. We extend relational technology to overcome such limitations. In addition to flat relations, we incorporate nested relations to efficiently store and access clustered complex objects. We support both hash and B-tree indexes to efficiently access objects through object identifiers. In addition to nested-loop join and sort-merge join, we provide hash join to efficiently process nonclustered complex objects in queries. We extend query optimization to process Object-Oriented queries including class hierarchies and method invocation. Note that such optimization is done not by the data management subsystem but by the object management subsystem. We provide user-defined manipulation and predicate functions directly evaluated on page buffers. Methods are compiled into them and efficiently processed. We devise object buffering in addition to page buffering and integrate these two schemes to evaluate queries. In a word, our approach is to provide an Object-Oriented model and language interface to an extensible database kernel [40], such as GENESIS [2] and EXODUS [5].

Fig. 5. System architecture.

Of course, there are alternatives to our extended relational approach to Object-Oriented database implementation. A pure relational approach such as Iris [30] has drawbacks as described above. Another approach uses WISS (Wisconsin Storage System) such as O2 [9], which provides record-based, single-relation operators. This makes it difficult for us to focus on query optimization based on set-oriented relational operators. In the extreme case, monolithic architectures could be considered in contrast to our layered approach. This would be less flexible to further tuning and extension. In this section, we will explain the function and implementation of the data management subsystem, and storage of objects and implementation of the object manipulation language.
3.1 Data Management Subsystem

3.1.1 Data Structures

Advanced applications of OODBs require a variety of indexes such as hash and B-tree indexes, and clustered and non-clustered indexes, and extended data dictionaries. Such indexes and data dictionaries are usually implemented as special data structures in relational database systems because of access efficiency. The conventional approach using special data structures makes the system less compact and less flexible to future extension. So the data management subsystem as a database kernel supports only relations (sequential, B-tree, hash, and inner relations) to allow the user of this subsystem to customize data dictionaries and indexes by using relations.

Only fixed-length and variable-length data are supported as field types of tuples by the data management subsystem. The data management subsystem makes no interpretation of field values except for TIDs and inner relations. Any type of data can be stored such as an array, a list, and a relation. Inner relations can be implemented as variable-length fields. Inner relations can have other inner relations as field values, so nested relations can be recursively defined. The length of a tuple must be less than the page size for efficient access and simple implementation. The length and number of fields in a tuple are subject to this limit.

The data management subsystem supports four types of relations as follows:

1) **SEQUENTIAL RELATIONS** have pages which are sequentially linked. Tuples are stored in the order of insertion. The location of inserted tuples is fixed, so an index can be created on sequential relations.

2) **B-TREE RELATIONS** have B-tree structures. Tuples are stored in the leaf pages in the order specified by user-defined order functions. This allows new access methods to be assimilated by supplying dedicated comparison and range functions. B-tree relations consist of key fields and nonkey fields. B-tree relations used as an index on sequential relations consist of several key fields and one TID field. This corresponds to a non-clustered index. B-tree relations which contain the whole data can be viewed as relations with a clustered index.

3) **HASH RELATIONS** use a dynamic hashing scheme called linear hashing with partial expansion [26], an extension to linear hashing. We choose this scheme because the space required to store data is proportional to the amount of data and the space utilization ratio is adjustable and high. Hash relations also consist of key fields and nonkey fields. The hash function is supplied by the user.

4) **INNER RELATIONS** for realizing nested relations are stored in variable-length fields of tuples. Tuples of inner relations are sequentially inserted. Nested relations can be recursively implemented by storing inner relations into fields of another inner relation. We provide nest and unnest operations for nested relations in addition to retrieval, insertion, deletion, and update. Retrieved inner relations can be operated as sequential relations. Update of inner relations can be done by retrieving inner relations, updating them as sequential relations and replacing old ones by new ones. We provide the functions interpreting the variable-length fields according to the nested relation schemes to operate on inner relations. Note that a theoretical basis for the nested relational model was provided by Kita-gawa and Kunii [24].

Tuple structures are uniform independently of relation types (see Fig. 6). The first two bytes of a tuple contains the tuple length. The tuple consists of fixed and variable parts. Fixed-length fields are stored in the fixed part. Variable-length fields are stored in the variable part. The offsets of the variable-length fields from the top of the tuple are stored in the fixed part. Any data can be accessed in a constant time although this tuple structure doesn’t allow null-value compression. Modification of the variable-length data can be done without affecting the fixed-length data.

![Tuple structure](image)

**Fig. 6. Tuple structure.**

TIDs, which can be stored in fixed-length fields, act as pointers to tuples. A variety of data structures can be implemented by using TIDs. For example, a non-clustered index can be implemented by defining an index key field and a TID field in B-tree or hash relations (see Fig. 7).

![Non-clustered index using a B-tree relation](image)

**Fig. 7. Non-clustered index using a B-tree relation.**

Access to fields must be efficiently processed since it is a frequent operation. We provide pointer arrays for field access (see Fig. 8). Each pointer points to the corresponding field in a tuple on page buffers. Simple tuple structures allow efficient construction of pointer arrays. One alternative is to copy field values to different areas. The alternative is good for data protection, but is rather time-consuming. Field pointer arrays are passed to user-defined functions such as manipulation and predicate functions for field access.

![Pointer array for field access](image)

**Fig. 8. Pointer array for field access.**
To efficiently access data, we move as few data as possible and fix tuples in buffers if possible. Internal sorting uses pointer arrays for tuples to be sorted (see Fig. 9). Such pointers are moved instead of tuples. Similarly, when a hash table is created for internal hashing, pointers to tuples are linked instead of tuples.

![Diagram of pointer array for internal sorting]

Fig. 9. Pointer array for internal sorting.

### 3.1.2 Hash-Based Processing

Set operations such as set-difference and duplicate elimination require OID-based access. Object-Oriented queries usually equi-joins based on OIDs. If either of two joined relations can be loaded into main memory, we can use the hash join method [39]. Even if neither of them can be loaded into main memory, the hash join method generally requires less CPU time and I/O times than the sort-based method. We adopted the hash-based method for equi-joins and set operations. Unlike Jasmine, other Object-Oriented systems such as ORION use nested-loop and sort-merge joins.

The internal hash join is used when either of two input relations for joins can be loaded into main memory. Recursion is not used in the internal hash join. Only one relation is partitioned into subrelations. The other relation is only scanned tuple by tuple. It is not necessary to load both of the relations entirely. We describe the outline of the algorithm. Only main memory is used during processing.

1) Determine which input relation is to be partitioned. Let the partitioned input relation be A.
2) Determine a partition number p and a hash function h.
3) Partition the relation A into p subrelations .
4) For each tuple b of the other relation B, compute k = h(key of b) and compare each tuple of A with k on the join key. When they match, make a new tuple from them and output it to the output relation C.

The external hash join is used when neither of two input relations can be loaded into main memory. The essential difference between the external hash join and the internal hash join is the use of recursion and the partitioning of both of the input relations. The outline of the algorithm is as follows:

1) Determine a partition number p and a hash function h.
2) Partition the relation A into p subrelations , and partition B into p subrelations .
3) For each , if either or can be entirely loaded into main memory, and are joined using the internal hash join. Otherwise, steps 1) through 3) are executed recursively.

### 3.1.3 User-Defined Functions

Methods are often specified in target and condition parts of Object-Oriented queries. Conventionally, applications retrieve all data and filter and manipulate them. This approach is rather inefficient because it requires extra data transfer and conversion between buffers and applications. System-defined comparators are also inefficient because they interpret any data according to data types. So we implement application-specific parts such as methods as user-defined functions and embed them into the data management subsystem. The user can specify application-specific parts as follows.

1) A predicate function specifies a retrieval condition of selection or join operators.
2) A manipulation function specifies operators on each tuple satisfying the predicate in selection or join.
3) An order function specifies the order used by sorting or B-tree relations.
4) A range function specifies a search condition of B-tree relations such as 
5) A static hash function is used by hash-based relational operators such as join, union, and difference.
6) A dynamic hash function is used by hash relations.

To separate application-specific parts by providing user-defined functions allows both flexible customization by the user and efficient execution by compiling. For example, the following predicate function filters tuples by using a simple condition with the system-defined comparison operator <:

```c
predicate1 (flag, OID, condition, age) {
  if (flag == MAIN) {
    if (age < 13) return true
    else return false
  }
}
```

Whether this predicate is invoked for PREprocessing, MAINprocessing, or POSTprocessing depends on the variable flag. Preprocessing and postprocessing are done only once while the main processing is done tuple-wise. Control is transferred to manipulation functions which manipulate filtered data only if the predicate function returns true. The predicate functions are compiled into operations on tuples in the buffer and are efficiently processed because no type is dynamically checked and no data is interpreted. The user-defined complex condition is also defined as a predicate function. If no predicate is explicitly supplied, control is always passed to manipulation functions.

Manipulation functions are compiled to operate directly on tuples in the buffer and are invoked only if predicate functions return true. For example, the following function operates on tuples of a relation by using a make-medical-certificate program and inserts the result into another relation tmp3:

```c
manipulate3 (flag, OID, doctor, name, disease) {
  if (flag == PRE) openinsert(tmp3);
  else if (flag == MAIN) {
    result = make-medical-certificate("19920826");
    insert(tmp3, result);
  }
  else if (flag == POST) closeinsert(tmp3);
}
```
In general, selection and join operators described below require predicate and manipulation functions. Data is thus efficiently filtered by predicate functions and operated on by manipulation functions.

3.1.4 Architecture

The data management subsystem has a layered architecture consisting of relational, tuple, and storage layers (see Fig. 10). All of these are open to the user. The data management provides neither query parser nor optimizer because they are rather high-level and application-dependent. It is just an executor of operators provided by the three layers.

```
relational layer
tuple layer
storage layer
```

Fig. 10. Architecture.

The relational layer provides functions which execute set operations as an extended relational algebra. For example,

1) `select ( rb, pb, mb)` extends selection of relational algebra. It has three parameters `rb`, `pb`, and `mb`. `rb` is the data block which specifies the source relation. `pb` and `mb` specify user-defined predicate and manipulation functions, respectively.

2) `hjoin(rb1, rb2, mb, hb)` performs equi-join of relations specifying `rb1` and `rb2`. `mb` is performed on each pair of tuples which match on join fields. This operation is based on a hash function specified by `hb`.

3) `join(rb1, rb2, pb, mb)` performs a general join of relations `rb1` and `rb2`.

4) `tjoin(rb1, rb2, tid, mb)` joins each tuple of `rb1` with a tuple of `rb2` pointed by its `tid` field and performs `mb` on such a pair of tuples.

5) `sort(rb1, rb2, ob)` sorts `rb1` and stores the result into `rb2`. The order function is specified by `ob`.

6) `unique(rb1, rb2, hb)` eliminates duplicates of `rb1` and stores the result into `rb2`. This operation is hash-based.

7) `nest(rb1, rb2, fid, hb)` generates a nested relation `rb2` from a flat relation `rb1` with fields specified by `fid`. This operation is also hash-based.

8) `unnest(rb1, rb2, fid)` generates a flat relation `rb2` from a nested relation `rb1`.

Functions of the tuple layer operate on four types of relations. The operators are as follows:

1) `scan` scans a relation sequentially and finds a tuple satisfying the specified predicate.
2) `raster` scans a relation sequentially fixing scanned pages on buffers. It is used in internal sorting or making internal hash tables.
3) `access` directly accesses a tuple satisfying the specified predicate.
4) `fetch`,
5) `delete`, and
6) `update` directly accesses, deletes, and updates a tuple specified by a given TID, respectively.

7) `insert` inserts a tuple or a group of fields.
8) `clear` deletes all tuples.
9) `alloc` constructs a field pointer array for the specified fields.

The functions of the storage layer include disk I/O, page buffers, transactions, concurrency control, and recovery. Disk I/O management includes allocation and deallocation of subdatabases (segments) and pages. A database consists of two types of subdatabases. One is a subdatabase which is permanent and recoverable. The other is a subdatabase which is used as a workspace for keeping temporary relations, and is only effective in a transaction. This is not recoverable. Subdatabases composed of a number of pages.

The storage layer supports variable-length pages sized `256K (i = 2, ..., 8)`, consisting of several 4KB physical pages, which form a virtually continuous page on buffers. We use the buddy system for buffer space allocation. The page length can be specified for each relation because multimedia data and inner relations may exceed 4KB.

We use concurrency control based on granularity, two-phase locking. Deadlock detection is done by examining a cycle in the Wait-For-Graph. One of the deadlocked transactions in the cycle in the graph is chosen as the victim for rollback. ORION uses deadlock detection based on timeouts. Our transaction recovery is based on shadow-paging for simplicity.

3.2 Object Management Subsystem

3.2.1 Object Storage

We efficiently store nested structures of objects by use of nested relations supported by the data management subsystem unlike other systems. Storage structures differ from instance to class. Translation of objects to relations is automatically done by the system. Information about the translation is held by classes.

All intrinsic instances of a class are stored in a relation, corresponding an instance to a tuple and an attribute to a field (see Fig. 11). Instances of different classes are stored in separate relations. Multiple-valued attributes such as Temperature are stored as a multiple-valued field, the simplest form of nested relations. We resolve attribute inheritance before storing instances. We treat inherited attributes such as Age and newly-defined attributes such as Weight uniformly. We store intrinsic instances of a superclass and those of a subclass in separate relations. If we instantiate or destroy intrinsic instances of a class, we don’t have to propagate any modification to its superclass or subclass.

```
<table>
<thead>
<tr>
<th>PERSON</th>
<th>DEPARTMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Medical</td>
</tr>
<tr>
<td>Age</td>
<td>Department002</td>
</tr>
<tr>
<td>Sex</td>
<td>Medical</td>
</tr>
<tr>
<td>Address</td>
<td>Patient007</td>
</tr>
<tr>
<td></td>
<td>Department005</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PATIENT</th>
<th>DOCTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Medical</td>
</tr>
<tr>
<td>Age</td>
<td>Medical</td>
</tr>
<tr>
<td>Address</td>
<td>Medical</td>
</tr>
<tr>
<td></td>
<td>Patient007</td>
</tr>
<tr>
<td></td>
<td>Department002</td>
</tr>
<tr>
<td></td>
<td>Department005</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DOCTOR</th>
<th>DEPARTMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Medical</td>
</tr>
<tr>
<td>Age</td>
<td>Medical</td>
</tr>
<tr>
<td>Address</td>
<td>Medical</td>
</tr>
<tr>
<td></td>
<td>Patient007</td>
</tr>
<tr>
<td></td>
<td>Department002</td>
</tr>
<tr>
<td></td>
<td>Department005</td>
</tr>
</tbody>
</table>
```

Fig. 11. Example of an instance table.
We store fixed-length strings and numbers in fixed-length fields and variable-length strings in variable-length fields. As for reference attributes such as Doctor we store only OIDs in fixed-length fields. This implements non-clustered complex objects. Non-clustered complex objects are needed by CAD applications where complex objects are created bottom-up, that is, component objects are reused. Of course, the user can enforce complex object integrity on non-clustered complex objects by attaching demons to methods of whole objects such as instantiate [17]. We correspond relation objects in attributes to inner relations of nested relations and their attributes to fields of the inner relations. Inherently, component objects of relation objects cannot exist without their whole objects. Clustered complex objects, implemented by relation objects, can be managed as a unit both logically and physically. Nested relations naturally realize clustering of complex objects although component objects must be hierarchically accessed from root objects.

The OID attribute also corresponds to a field. An OID consists of a database id-number, a class id-number, and an instance id-number, so an OID alone can inform us of its database and class information directly. Unlike the OID of O2 [9], the OID of Jasmine is logical in that it contains no physical addresses in secondary memory. No OIDs need to be changed even at data reorganization although OIDs need to be changed if objects are migrated to another class. The user can choose among sequential, B-tree, and hash relations as instance storage. The user can attach indexes to attributes.

As classes are instances of the system-defined CLASS in Jasmine, we store all classes in one nested relation and facilitate efficient associativve access of class objects. Basically we correspond one class object to one tuple. Since attribute categorization, such as enumerated and procedural, is common to all classes and attributes have a fixed set of facets, we store enumerated and procedural attributes in different inner relations and facets in the fields of the inner relations. Procedural attribute (method) definitions are also stored in relations and are retrieved and utilized during query optimization. The system-defined attributes such as Super are stored in separate fields (see Fig. 12). To store heterogeneous classes in one relation makes set-oriented access to them efficient.

### 3.2.2 Set Oriented Access Support

We compile both set-oriented access and singleton access to do early binding and reduce run-time overhead. The Jasmine compiler is implemented using a C compiler. The application programs written in Jasmine/C are precompiled to C programs, which are compiled and linked with the run-time support library. Preprocessing is used to take maximum advantage of portability and code optimization of the C compiler. An interactive query needed by application development is processed by the interpreter, not by the compiler.

Application programs are translated into C in three phases: query graph generation, access plan generation, and code generation. The query graph generation phase makes a query graph corresponding to the user query by referencing the object model. The query graph is a subgraph of the object model annotated with the target and condition information. For example, consider the query:

```
PATIENT.made-medical-certificates(“19920826”) where PATIENT.Age < 13 and PATIENT.Doctor.Name == “Dr. No”
```

This makes the following query graph:

```
CHILD-PATIENT(made-medical-certificates(“19920826”), Age < 13, DOCTOR(Name == “Dr. No”))
```

Note that the user query containing incomplete knowledge access is transformed into a more efficient one during this phase. In Jasmine, the user can form a query by specifying a general class such as PATIENT instead of the specific class CHILD-PATIENT. Then it is necessary to restrict the general class to its appropriate subclass.

Then the access plan generation phase translates the query graph into an optimal plan of a series of extended relational operators using the object model information such as statistics, access methods, and mapping from class to relation. This phase uses rule-based optimization. Rules are grouped into sets of rules called rulesets [15] according to patterns of queries. This increases the modularity of rules. Rulesets can be more efficiently processed than flat rules because of this modularity. They are also easier to maintain and extend. Object-Oriented query optimization is fully described later. Here we just illustrate the overall flow of query processing. Object-Oriented features such as complex objects, class hierarchies, and methods (procedural attributes) constitute patterns of rulesets. Their occurrences in queries invoke associated rulesets. For example, the object expression is processed by different rulesets depending on whether it contains a procedural attribute or not.

First, the case where object expressions contain no procedural attributes is considered. In general, the object expression generates equi-joins between instance relations corresponding to a functional join. In addition, the conditions generate selections and explicit joins. For equi-join, a predicate function which joins two instance relations by an attribute field of one relation and an OID field of the other is generated. For selection, a condition concerning one instance relation is generated as a predicate function of a select operator. For an explicit join, a join predicate function which may contain non-equijoin is generated. Manipulation functions are generated to project fields for later operation.
The query results in a set of OIDs, values, or tuples. A query against a nonleaf class evaluates to a relation containing OIDs of instances of several classes. As OIDs have the same structure for all objects, they can be stored in one relation. Each scan returns an object, by scanning the result relation and then selecting the base instance relation by the OID. At that time, if an object is already on core, it is used.

Usually, if a selection predicate of the sequential relation can use an index, selection by index is chosen which selects a B-tree relation for the index by the key condition, and sorts the result relation containing TIDs and then joins the result relation and the original sequential relation by using TIDs. The rest of the selection condition is evaluated at the same time. For B-tree and hash relations, if a predicate concerns the key fields, key-based searching is done. Notice that if a whole relation of any type is small enough to be contained within one page, sequential access is chosen.

If one of two relations being joined is small enough to be contained within a page and the join key is indexed by the other relation, tuple substitution is chosen. If one of two relations is contained within a page and no index is provided, nested loop is chosen. Otherwise, hash join is chosen. For B-tree and hash relations, the join is similarly processed. In case of a join of several relations, the order of join is dynamically determined by the size of the intermediate result relations. First, we choose the smallest relation and the second smallest one among relations to be joined. Then we join them to obtain an expectedly small relation as a result. We add the result to relations to be joined and repeat this process. This dynamic scheme based on exact relation sizes is expected to be more efficient than static schemes based on database statistics.

Next, consider the case where object expressions contain procedural attributes. User procedural attributes appearing in the target part are translated into manipulation functions. Procedural attributes in conditions are translated into predicate functions. For example, the above query graph generates the relational operator sequence and the predicate and manipulation functions as follows:

```sql
select (DOCTOR, predicate1, manipulate1);
select (CHILD-PATIENT, predicate2, manipulate2);
if (within-page(tmp1) || within-page(tmp2))
  join(tmp1, tmp2, predicate3, manipulate3);
else hjoin(tmp1, tmp2, predicate3, manipulate3, hashfune);

predicate1 (flag, OID, name)
  { if (flag == MAIN)
    { if (name == "Dr. No") return true
      else return false }
  manipulate1(flag, OID)
    { if (flag == PRE) openinsert(tmp1);
      if (flag == MAIN) insert(tmp1, OID);
      else if (flag == POST) closeinsert(tmp1); }

predicate2 (flag, OID, condition, age)
  { if (flag == MAIN)
    { if (age < 13) return true
      else return false }
  manipulate2 (flag, OID, doctor, name, disease)
    { if (flag == PRE) openinsert(tmp2);
      if (flag == MAIN)
        [Acertificate = make-medical
         -certificate("19920826");
         insert(tmp2, doctor, Acertificate); ]
      else if (flag == POST) closeinsert(tmp2); }

else if (flag == POST) closeinsert(tmp2);

predicate3 (flag, OID, doctor)
  { if (flag == MAIN)
    { if (OID == doctor) return true
      else return false }
  manipulate3 (flag, Acertificate)
    { if (flag == PRE) openinsert(tmp3);
      else if (flag == MAIN)
        insert(tmp3, Acertificate);
      else if (flag == POST) closeinsert(tmp3); }
```

If an index on Name of DOCTOR is available, the first select will be replaced by the sequence select-sort-tjoin (selection by index). The system-defined procedural attributes such as aggregate functions and update functions are also translated into manipulation functions. In particular, the update functions are translated into relational update operators. If demons are defined and the option is specified, they are integrated into manipulation and predicate functions. Lastly, the C codes for the given query are generated to feed the C compiler.

### 3.2.3 Object-Oriented Query Optimization

The features of query optimization in an Object-Oriented database are different from those of query optimization in a relational database because multiple-valued attributes, implicit joins, procedural attributes (methods), and nonleaf classes in a class hierarchy are specified in a query. First, we describe processing of multiple-valued attributes. As for non-clustered complex objects, reference attributes contain only OIDs and multiple-valued attributes contain only elements of a set. Then multiple-valued attributes contain a set of OIDs or values. Since only sequential access is supported for inner relations of nested relations, multiple-valued attributes are unnested into flat relations and are optimized conventionally except for application of aggregate functions. As for clustered complex objects implemented by nested relations, predicate and manipulation functions of inner relations of nested relations are nested into those of outer relations of the nested relations. They are recursively evaluated from outer relations to inner relations.

Next, we describe implicit joins of relations generated by object expressions such as DOCTOR.Patient.Age. If there is no available index on the OID field of the relation for the class of the attribute (e.g., PATIENT), the join is processed by hash joins. The order of more than one joins is dynamically determined by the size of the intermediate result relations. If there is an index available on the OID, the join is processed by TID joins. In case of several joins, they are processed from left to right in the object expression. Section predicates, if any, are evaluated during join processing. Note that there are methods for precomputing joins. For example, to process the query (DOCTOR.Patient.Age where DOCTOR.Patient.Age > 30), an index with Age as a key value and the OID of DOCTOR as a pointer value is created. Other systems such as ORION use this approach. However, it is rather difficult to maintain such an index properly.

We describe how to process queries containing nonleaf classes in a class hierarchy. We assume that PATIENT has ADULT-PATIENT and CHILD-PATIENT as subclasses. Consider the following examples,

**QUERY 21.** PATIENT.Name where PATIENT.Age > 12 and PATIENT.Age < 20
QUERY 22. DEPARTMENT.Doctor.Patient.Name where DEPARTMENT.Name == "pediatrics"

For Query 21, the system generates two subqueries:

\[
\text{result} = \text{ADULT-PATIENT.Name where ADULT-PATIENT.Age < 20}
\]

\[
\text{result} + \text{CHILD-PATIENT.Name where CHILD-PATIENT.Age > 12}
\]

The two query results are inserted into the same output relation.

For Query 22, the join of DEPARTMENT and DOCTOR is processed first. During the join processing, the intermediate output relations are switched according to the class of the OID for DEPARTMENT.Doctor. Patient. The class can be determined just by looking at the OID. The pseudo queries are as follows:

\[
\text{adult-intermediate} = \text{DEPARTMENT.Doctor.Patient where DEPARTMENT.Name == "pediatrics" and DEPARTMENT.Doctor.Patient.Class} = \text{ADULT-PATIENT}
\]

\[
\text{child-intermediate} = \text{DEPARTMENT.Doctor.Patient where DEPARTMENT.Name == "pediatrics" and DEPARTMENT.Doctor.Patient.Class} = \text{CHILD-PATIENT}
\]

The switching is done during a single join operation. The code for the switching is translated into the manipulation function of the join operator. Then a pair of adult-intermediate and ADULT-PATIENT and a pair of child-intermediate and CHILD-PATIENT are joined, and the results are merged. As described above, the intermediate result of selection or join operations is switched to separate relations containing only OIDs relevant to successive joins. This can establish optimal preconditions for the joins by avoiding unnecessary search.

Classes (e.g., PERSON, DOCTOR, and PATIENT) in a class hierarchy share inherited attributes such as Age. Basically there are two methods for creating indexes on classes in a class hierarchy. One method is to create only one index on a whole class hierarchy, called a class-hierarchy index. The other is to create a separate index, called a single-class index, on each class. The index uses single-class indexes. Other systems such as ORION and O2 use class-hierarchy indexes. The class-hierarchy index has an advantage that the total size of index pages and the total number of accessed index pages are smaller than those of the single-class index. However, it is not always optimal when a class hierarchy is partially specified in a query. Moreover, it is rather difficult to maintain such class-hierarchy indexes.

In some cases, semantic information such as categorization can be used to specialize nonleaf classes to specific ones. When a condition on the categorization attribute such as (Age < 7) is specified in a query containing a nonleaf class PATIENT, if the condition matches one of the categorization conditions of partition classes (Age < 18 for CHILD-PATIENT), the nonleaf class (PATIENT) is specialized into the subclass (CHILD-PATIENT) with the matched categorization condition. This can reduce the size of the search space for the query.

The system translates methods such as make-medical-certificate of PATIENT specified in a query into the manipulation and predicate functions of selection or join operators, and processes them on page buffers, which avoids unnecessary data transfer between page buffers and application programs. Methods defined by a query such as internal of DEPARTMENT is expanded into the outer query. To this end, the source codes and compiled codes for methods and demons are stored as program objects in databases. They are retrieved and compiled during query optimization. To store programs in databases makes the integration of query and programming facilities more elegant than to store them in ordinary program files. Polymorphic methods are translated into their corresponding implementation functions.

3.2.4 Object Buffering

Singleton access is also compiled into C programs, which are compiled and linked with the run-time support library. First, run-time support will be described. The first access of an object fetches the object from secondary memory to the page buffer. Then the object is cached in the active object table (AOT), a dedicated internal hash table for object buffering (see Fig. 13).

![Fig. 13. AOT structure.](image)

The primary role of AOT is to efficiently look up objects. When an instance is referenced through its OID for the first time, the instance is hashed by its OID as a hash key. A hash entry (an object descriptor) and an in-memory instance data structure is created. The hash entry points to the instance data structure. If the instance is referenced through its OID by other instances resident in AOT, the OID is mapped to the pointer to the hash entry through the AOT. The pointer can be cached into the attribute of the referencing instance since an OID is longer than a physical pointer. Later, the instance can be directly accessed by the pointer without hashing.

Another important role is to maintain the status flags for update of objects. When an object is newly created or updated, the status flag in the hash entry for the object is set to create or update. When a transaction is committed, the object with the status create or update is modified or added into the page buffers. When an object is destroyed, the corresponding in-memory data structure is deallocated and the status flag is changed to destroy. Later, if the destroyed object is referenced, the validity of reference is checked and an exception handler is invoked. This can support referential integrity. When a transaction is committed, the object is destroyed in databases.

When objects fill up AOT, extraneous objects are swapped out. Such an object is flushed to the page buffers and the in-memory instance data structure is deallo-
cated and the status flag in the hash entry is set to free. When the object is referenced again, the object is directly fetched from databases to AOT through its TID cached in the hash entry.

The object management subsystem requires AOT, that is, object buffers in addition to page buffers of the data management subsystem for the following reason. In general, buffers are directly associated with patterns of access of objects. Page buffers have structures suitable for access of different instances of the same class. AOT have structures suitable for access of correlated instances of different classes. Advanced applications such as CAD have combinations of two such patterns. This necessitates a dual buffer scheme consisting of page buffers and object buffers, not a single buffer scheme, which would contain unnecessary objects and decrease memory utilization.

The dual buffer approach, however, makes the same object appear in different formats in different buffers at the same time, so we have to maintain internal consistency between two objects denoting the same entity. Currently, we first write back updated or newly created instances from AOT to page buffers in query evaluation. Then we evaluate a query against page buffers. An alternative is to devise different search mechanisms for different buffers and evaluate the same query on different buffers and integrate the results, which would make the system less compact.

Basically there are two methods for query evaluation using object buffers and page buffers as follows.

**Single-buffer evaluation method:**
1) The instances newly created or updated associated with the classes specified by the query are searched in the object buffers.
2) They are flushed from the object buffers to the page buffers.
3) The query is evaluated against the page buffers.

**Dual-buffer evaluation method:**
1) The query is evaluated against the object buffers.
2) The same query is evaluated against the page buffers.
3) The two results are merged into one.

Jasmine adopts the single-buffer evaluation method while ORION adopts a more sophisticated version of the dual-buffer evaluation method. The single-buffer evaluation method needs to transfer objects from the object buffers to the page buffers. However, the single-buffer evaluation method eliminates the need for dual evaluation programs and makes the system small and processing simple in contrast to the dual-buffer evaluation method. Anyway, the combinational use of object buffers and page buffers can support the integration of programming and query facilities at an architecture level.

## 4 Application

This section describes an Object-Oriented database approach to engineering applications [18]. First, we discuss design data management, intelligent CAD support, and engineering information management as requirements for support of engineering tasks. Then we explain our OODB solutions to the above requirements, which are embodied in a prototype intelligent CAD system, called HyperCAD [18].

### 4.1 Issues for Engineering Tasks

Only a fraction of engineering tasks including planning, design, experiments, and manufacturing are supported by computers. Thus, conventional CAD systems handle only part of primary design data, focusing on drawing processes. Planning and conceptual design prior to drawing are not supported. Information relevant to the entire engineering task is not managed. In order to increase the total productivity and reliability of engineering tasks, we should support these processes and manage engineering information by computers.

Both structural and behavioral representation of primary design data is mandatory in design processes. As structural aspects, we must represent relationships between design objects and their components and relationships between similar design objects. Direct support of these relationships is a basic requirement for design data management. As behavioral aspects, we must describe design constraints which hold between design objects. We must also describe design methods to generate candidate solutions which satisfy such design constraints. We call this requirement intelligent CAD support.

We must create, retrieve, and manipulate engineering information, such as engineering documents, relevant to primary design data throughout the engineering processes. Even designers must spend most of their time by doing such tasks. Engineering documents consist of large-scale data of various types such as numbers, character strings, texts, graphics, and images. So a variety of data types must be integrated. Heterogeneous media data must be treated uniformly and flexibly. Relationships between heterogeneous media data must also be managed. As engineering documents will be subject to revision, we must maintain consistency constraints between related media data. These requirements are not comprehensive (for example, support for cooperative work by a group of designers is also important and is described in Section 5), but vital for support of engineering tasks. In the rest of this section, we describe our OODB approaches to each of the above requirements.

### 4.2 Design Data Management

Direct representation of component relationships between design objects and generalization relationships between similar design objects is needed by design data management. These facilities can be implemented by complex object modeling and class hierarchies of OODB. For example, we model design objects using Jasmine as follows. The attribute Part of the class UNIT denotes that units consist of multiple components (see Fig. 14). The attribute Super of the class PISTON, which supports generalization, denotes that pistons inherit attributes of components. Thus, OODBs can reduce redundancy of schemas and represent component relationships directly.
The semantics of component relationships vary from application to application while those of generalization relationships are the same. Therefore, Jasmine has no built-in semantics of component relationships. Part is just a user-defined attribute for Jasmine. Instead, the user can provide application-specific semantics to attributes and methods as demons in Jasmine. For example, there is a consistency constraint such that components can only exist when associated units exist (this constraint doesn’t necessarily hold in bottom-up design where components are prior to units). This constraint can be implemented by specifying programs instantiating components as an after demon of the instantiate method of the class UNIT and programs destroying components as a before demon of the destroy method of UNIT. As another example, such a constraint that the mass of a unit is the sum of the mass of its components can be implemented as if-needed demon attached to the attribute Mass of UNIT (see Fig. 14). This can maintain the mass of units in a consistent state. Jasmine demons manage user-specified consistency constraints.

OODBs can represent dynamic characteristics of design data as methods. For example, the method MaxBendingStress calculates the maximum bending stress imposed on a piston head (see Fig. 14). OODBs can not only define such application-specific functions but can provide generic operations such as retrieval and update by encapsulating them as system-defined methods of general objects. Flexible retrieval of design data is important in engineering applications. Jasmine allows the user to retrieve data from complex, large-scale data sets by providing a set-oriented query language (see Fig. 15). Jasmine also facilitates advanced flexible searching by allowing the user to specify methods in queries. Generally, OODBs contribute to increase productivity in application development as the user can make application programs by combining methods and queries without considering object persistency.

UNIT where UNIT . Mass < 20.0 and UNIT . Part . Name = "conrod"
(retrieve units which are under 20 units of mass and have a part named "conrod")

Fig. 15. A query.

4.3 Intelligent CAD Support

Even sophisticated CAD systems supporting drawing in design do not sufficiently improve the total productivity of design processes. The designer should be able to represent a collection of design constraints and various design methods for generating candidate solutions to design constraints. As constraints are rarely satisfied at the beginning, the designer should be able to specify what to do when the constraints are unsatisfactory. We call this ability intelligent CAD support. Here we describe a Jasmine approach to intelligent CAD support.

Special objects, called design goals, for management of design constraints are provided in addition to design objects. The user can describe design constraints, design methods, and advice for constraint failures in design goal objects. The user can specify a collection of constraints on attributes of design objects as design constraints. The user can choose among database (table) retrieval, calculation based on other attributes, generation, and user input as design methods. The user can invoke other design goals or its own. These descriptions constitute a part of design knowledge.

![Diagram](image)

Fig. 16. Design goal: (a) a dependency network, (b) goal definition.

Solutions satisfying the constraints are searched for, based on a network consisting of design goals and dependency relationships between them. Such a dependency network is created prior to finding the solutions. During that time, error checks are done. Fig. 16a depicts a part of a dependency network. Fig. 16b describes a part of design goal definition for determining the value of the attribute PistonHeadThickness of PISTON. The constraints in the design goal describe what this attribute must satisfy. The design method generates candidate values from the initial value and increase value. The advice for failure of the third constraint sends its own goal a message to increase in value or sends a message to decrease in value to the goal for Diameter of PISTON.

We take a so-called knowledge-based backtrack approach to constraint failure in that the user controls the backtracking directly. In contrast, in most constraint programming languages [27], the user only describes constraints and leaves constraint satisfaction to the system. That is, the satisfaction mechanism is hidden from the user. Such a black box scheme is inappropriate for design because techniques of constraint satisfaction are a part of design knowledge. So
we provide a scheme such as design goals where the user can describe design knowledge directly and the system can interpret the knowledge. Although pure OODBs provide no direct support for constraint management, we can implement constraint management just by adding objects for the function.

The system creates plans or graphics associated with design objects, based on design solutions inferred from the constraints. As with graphics, there are also geometrical constraints such as the connection of components. We can make graphics of possible states by resolving such geometrical constraints. By repeating this process for each graphic at different times, we can make an animation series to facilitate primitive simulation in conceptual design. We call this approach constraint-based animation. For example, Fig. 17a depicts a part of the geometrical constraints and Fig. 17b depicts an animation still resolving such constraints. Note that graphics data must be updated if associated design objects are updated. This can be implemented by supplying if-updated demons to attributes of design objects.

ConrodGeo.\text{x} = \text{CrankGeo.\text{CrankHead1.\text{center.x}}}
ConrodGeo.\text{y} = \text{CrankGeo.\text{CrankHead1.\text{center.y}} + CrankGeo.\text{CrankArmThickness} + (\text{CrankGeo.\text{CrankPinLength}-ConrodGeo.\text{ConrodThickness})/2}}

Fig. 17. Constraint-based animation: (a) a geometric constraint, (b) an animation still.

4.4 Engineering Information Management

Engineering documents consist of various data, such as texts, figures, tables, and relationships between them, so a hypermedia approach seems promising for engineering information management. Hypermedia systems, such as NoteCards [12], have simple data models consisting of nodes and links that allow data retrieval by link traversal. However, most current hypermedia systems, if they were applied to complex, large-scale applications such as engineering tasks, have problems with complex object modeling, efficient management of large-scale media data, flexible searching facilities, programming facilities, and change management related to application evolution. Here we describe an OODB approach to the basic functions and advanced issues of hypermedia in general.

OODBs can accommodate media data easily. Basic media such as graphics, images, and text are defined as objects. Long data is stored in attributes of byte strings. In Jasmine, the user can optimally adjust the maximum length of data contained within one page by specifying the page size of each object. Operations on media are defined as object methods. Heterogeneous media data can be operated on uniformly using polymorphism.

Presentation of complex objects needs simultaneous presentation of various types of attribute values. To this, we must define composite media data as complex objects. Sometimes, objects contained in other text objects need to be accessed, such as design goals associated with design guidebooks (see Fig. 18). Conventionally, this is implemented by managing their OIDs and display areas as described above. In contrast, we have taken an approach where we create a table consisting of keywords appearing within texts and OIDs of associated objects in addition to texts. If the user drags a keyword string in a presented text, the system selects the OID of the associated object by searching the table. This approach eliminates the need for recalculating display areas associated with updated texts and for managing multiple entries for objects recurring in texts.

<table>
<thead>
<tr>
<th>GoalName</th>
<th>CrankRadius</th>
</tr>
</thead>
<tbody>
<tr>
<td>DesignMethod</td>
<td>calculate(Stroke/2)</td>
</tr>
<tr>
<td>Constraint</td>
<td></td>
</tr>
</tbody>
</table>
| $l = x \times r$ 
| where |
| $l$ : the pitch 
| $x$ : magnification(3.6–4.2) 
| $r$ : crank radius |

Fig. 18. Traversal of objects contained within text.

We take an OODB approach to current hypermedia issues [17]. Direct manipulation of hypermedia is not enough for complex, large-scale engineering tasks although it is extremely useful. The user should be able to do flexible searching using complex conditions and procedures. In Jasmine, the user can do advanced searching such as structural and content searches by invoking dedicated methods in queries. To manage change associated with application evolution, we allow the user to specify queries within methods in order to define virtual nodes and links. For example, Fig. 19a defines a virtual link. Fig. 19b depicts use of the virtual link through an end-user interface. The query is also represented as an object. Ad hoc queries can be interactively interpreted while routine queries can be compiled and efficiently executed. For complex object modeling, the user can maintain consistency constraints between heterogeneous media objects by using demons. For example, if the value of an attribute of a design object, such as PistonHeadThickness, is updated, the associated graphic objects, such as plans, are updated by an if-updated demon attached to the attribute.
4.5 Summary

We discussed the requirements for supporting engineering tasks and described our OODB approach to them, focusing on management of constraints such as description, satisfaction, and maintenance. We call this the constraint Object-Oriented approach. This has been realized in a prototype intelligent CAD system HyperCAD which consists of more than one hundred classes defined by Jasmine. Constraint management is realized by rules in other database systems such as POSTGRES [38] and Starburst [29]. They provide rule facilities as an extension to relational databases in order to check integrity constraints and propagate updates. Such functionality is given by the trigger mechanism such as demons in Jasmine. Our approach described here focuses more on how to populate databases by satisfying constraints. We provide design goals, or constraint objects, for the purpose.

We compare HyperCAD with other relevant works. KRIKYS [7] is similar to HyperCAD/Jasmine in that KRIKYS provides a knowledge base management system for the construction of advanced CAD systems. It, however, lacks a facility for managing constraints.

Like HyperCAD, VT/SALT [32], AIR-CYL/DSPL [4], and PRIDE [33] [34] take constraint-based approaches. PRIDE enables the user to explore different alternatives simultaneously by maintaining multiple design contexts while HyperCAD, VT/SALT, and AIR-CYL/DSPL provide design alternatives sequentially. VT/SALT, AIR-CYL/DSPL, and PRIDE are largely different from HyperCAD in that these three systems lack facilities for abstraction mechanisms, behavioral modeling, and geometric constraint management.

5 EXTENSION-SCHEMA TRANSLATION BY VIEW

5.1 Schema Translation

In a distributed and coordinated environment such as concurrent engineering, semantically identical data are often represented by different schemas of an Object-Oriented database at different sites. However, users want to look at schemas defined by others from their own points of view. This necessitates schema translation. There is an approach to this, called a global schema approach [36], which considers a global schema to integrate all local schemas at different sites. However, it is not necessarily possible to integrate heterogeneous schemas when there are many database schemas. So schema translation between relevant sites is more practical. This is called a federated database approach [13].

In schema translation of an Object-Oriented database, there are issues where the semantically same attributes (enumerated attributes or properties) have different names, different representations, different structures, or missing data depending on the site. The semantically same methods (procedural attributes) may also have different names, different interfaces, and different implementations. Different attributes or methods may even share the same name. Our goals are to allow users to reference databases defined at other sites through their own views, and to reference as many databases as possible at the same time. We describe that schema integration by views can be realized by extending the kernel classes of Jasmine. We focus on resolution of semantic heterogeneity within homogeneous systems (i.e., Jasmine). Note that so-called heterogeneous databases also need to resolve heterogeneities in systems and data models.

5.2 View Mechanism

We take an Object-Oriented database view approach to schema translation [20]. View definition is similar to class definition. A view in an Object-Oriented database allows the user to define new classes by modifying the definitions of existing classes at his own and other sites and by restricting a set of existing objects. The user can choose among attributes or derive new attributes by combining existing ones. The user can also define new methods for the view class. Note that views are different from inheritance in that views allow the user to neglect attributes and modify attributes as well as to add attributes. We describe the view mechanism as an extension to Jasmine. A class where a user defines a view is called a view class. A view class specifies original sites and classes, attributes, methods, data translation rules, and data selection conditions. The user can specify multiple sites at the same time. The original class specifications include class names and their site category such as own, other, or null (both). Attributes of a view class are defined as usual. View methods are defined like class methods. In data translation rules, the user can derive new attributes by operating on attributes of original classes. The user can also specify methods of the original class in the data translation rules. The data selection condition is specified by the Jasmine query language. For example, in Fig. 20, the view SeriousPatient defines a subset of the base class PATIENT at SITE1 by specifying a selection condition. Here, the attributes Name, Doctor, and Disease are inherited, the other attributes are deleted and the attribute Adviser is derived.
Attributes of a view are derived according to data translation rules, if any, otherwise they are copied from corresponding attributes of the original classes. If multiple classes at different sites are specified and methods are specified in the data translation rules, the methods usually have different implementations depending on the site. This is a use of polymorphism in views. For example, to integrate geographically distributed hospitals, the translation rule for the attribute Age of PersonView at SITE1 contains the method make_age, which has different implementations depending on SITE1, SITE2, and SITE3 (see Fig. 21). However, they have the same interface and are invoked at the same time and merged into Age of PersonView. This view constructs a superset of the three classes.

5.3 Implementation

View class definition and view class instantiation are distinct operations. A view class is defined at the site where the view is used, but is actually instantiated at the site where the origin-
5.4 Summary

Our OODB approach to schema translation in a distributed and coordinated environment focuses on view mechanisms, which we have described as an extension to our prototype OODB, Jasmine. We compare our approach with relevant works. A global schema approach called superview [36] considers a virtual database as a global database and creates the virtual database and queries against it. We don’t take the global schema assumption. Like our approach, MRDSM [28] is based on a federated database approach. The data definition language defines views over the relational databases being integrated. However, it is difficult to extend this approach to other data models. DomainMatch [8] focuses on domain mismatch problems. This approach uses virtual attributes and domain mappings. This approach is independent of the global schema and federated database approaches, but it does not apply to models other than relational models. Pegasus [1] uses an Object-Oriented approach to provide a uniform framework for heterogeneous databases. To realize the functionality, Pegasus extends the language of its basic Object-Oriented database system while our approach extends the object model of Jasmine by introducing view objects.

6 Conclusion

First, in this paper, we described a prototype Object-Oriented DBMS called Jasmine, focusing on the implementation of its Object-Oriented features. Jasmine shares a lot of functionality with other Object-Oriented database systems. However, Jasmine has the following features which differentiate it from other systems. Jasmine provides a powerful query language which allows users to specify complex objects, class hierarchies, and methods in queries. Jasmine optimizes such Object-Oriented queries by using hash joins, B-tree and hash indexes, and semantic information. Individual object access is evaluated on object buffers. Jasmine extends relational database technology. Jasmine provides nested relations to efficiently manage complex objects and provides user-defined functions evaluated on page buffers to efficiently process method invocation in queries. Jasmine provides a view facility for schema integration and a constraint management facility including integrity constraints, triggers and rules. We compare Jasmine with current commercial Object-Oriented database systems and research prototypes as follows.

ObjectStore [25] can make C++ objects databases. Any data in C++ programs can be persistent. A query, as an extension to C++, consists of a collection of objects and C++ Boolean expressions. The result of a query is restricted to a subset of the single collection being queried while a Jasmine query can combine more than one class to newly create instances. Jasmine uses hash joins instead of path indexes to process object expressions.

GemStone [3] originates from the attempt to make Smalltalk-80 programs databases. The GemStone data model is based on Smalltalk-80 and supports only single inheritance while Jasmine supports multiple inheritance. In addition to C, C++, Smalltalk-80 interfaces, GemStone provides a programming interface called OPAL. GemStone distinguishes between a class and a collection of objects. A query expressed by OPAL is formulated against a single collection of objects. A Jasmine query is formulated against classes, allowing explicit joins.

ORION [23] supports a variety of functions, such as multiple inheritance, composite objects, versions, queries, and schema evolution. ORION is built in Lisp on a secondary storage system which provides facilities for segment and page management. ORION provides a programming interface to an Object-Oriented extension of Lisp. A query returns a collection of instances of a single class while a Jasmine query can generate instances combining more than one class. Mapping object identifiers to pointers is done by extensible hashing. A query with attributes of nonleaf classes is processed by use of a class-hierarchy index unlike Jasmine. ORION evaluates a query against the object and page buffers and merges the results while Jasmine uses the single evaluation scheme. ORION uses sort-merge joins while Jasmine uses hash joins.

In O2 [9], an object contains a value, a list, a set, and a tuple as an attribute value. O2 is used through an Object-Oriented extension of C called CO2. The query language is defined rather formally. The query retrieves and composes a list, a set, and a tuple. O2 is implemented on top of WISS (Wisconsin Storage System) in C. WISS provides persistence, disk management, and concurrency control for flat records. Unlike Jasmine, O2 uses physical identifiers of WISS records as object identifiers. Like ORION, O2 adopts a dual buffer management scheme. Like Jasmine, O2 uses a hash table to manage in-memory objects, but unlike Jasmine, O2 uses a class-hierarchy index to process queries against nonleaf classes.

In IRIS [30], based on the DAPLEX functional model, properties or methods defined by a class are represented as functions on the class. Functions are stored or derived from other functions. IRIS supports multiple inheritance, versions, schema evolution, and queries. Query optimization is done by rule bases. Unlike Jasmine, IRIS is implemented on a relational storage system which supports only flat relations. IRIS has C and Lisp interfaces, but supports no integration with Object-Oriented programming languages while Jasmine does.

Next we discussed an Object-Oriented database approach to engineering as an advanced application. Then we described schema translation by view as an extension to Jasmine. OODBs have just been developed, so there are very few reports of real applications. We would like to apply our OODB Jasmine to various real-world problems, not only to verify the validity of our approach but to also give feedback to Jasmine from the experiences.

Our future plans include research on technical issues asso-
associated with exploratory aspects of advanced applications such as design: The incorporation of version management, constraint management, and view management in a heterogeneous environment. Version management is mandatory for exploratory applications, but concepts of versions differ from application to application. It is important to propose generic concepts of versions from which specific versions can be derived and to include both instance and class versioning. To explore design alternatives and propagate updates, we must incorporate generalized constraint management including constraint satisfaction rules and composite events [22]. To support cooperative exploration in a heterogeneous environment consisting of relational and Object-Oriented systems, we must provide more advanced view support which allows the user to look at schemas defined by other users in the system (e.g., relational systems) differently from the original ones as if they were Object-Oriented.

REFERENCES

Hiroshi Ishikawa received the BS and PhD degrees in computer science from the University of Tokyo, Tokyo, Japan. He joined Fujitsu Laboratories Ltd., in 1979. He is the chief architect of an object-oriented database system, Jasmine. His research interests include object-oriented databases, multimedia databases, distributed databases, and active databases. He has published actively in international, refereed journals and conferences on object-oriented databases, active databases, multimedia systems, and knowledge-based systems. He has authored a book Object-Oriented Database System (Springer-Verlag). He received the Sakai Memorial Distinguished Award from the Information Processing Society of Japan in 1984.

Dr. Ishikawa is a member of the IEEE, the ACM, Information Processing Society of Japan, and the Institute of Electronics, Information and Communication Engineers.

Yasuo Yamane received the BS degree in basic science in 1979 and the MS degree in computer mathematics in 1981 from the University of Tokyo, Tokyo, Japan. He joined Fujitsu Laboratories Ltd. in 1981. He worked on the design and implementation of some prototypes such as a distributed database system RDB/DY, a parallel database processing system, a high-speed extended relational database engine XDE, and an object-oriented database system Jasmine. He has been developing a commercial Object-Oriented database system ODBII at Fujitsu Ltd.

Mr. Yamane is a member of the Information Processing Society of Japan, and the Institute of Electronics, Information, and Communication Engineers.

Yoshio Izumida received the BS degree in 1975 and MS degree in 1977 in electronic engineering from Yokohama National University, Yokohama, Japan. He joined Fujitsu Laboratories Ltd. in 1977 and is now a manager of Database Special Project Department of Fujitsu Limited. His current research interests include object-oriented databases, multimedia databases, and parallel processing.

Mr. Izumida is a member of the IEEE, the ACM, Information Processing Society of Japan, and the Institute of Electronics, Information and Communication Engineers.

Nobuaki Kawato received the BS degree in 1971, MS degree in 1973, and PhD degree in 1976 in electronic engineering from the University of Tokyo, Tokyo, Japan. He has been with Fujitsu Laboratories Ltd., since 1976 and now is a manager of the Computer Network Systems Laboratory. His research activities have been in various areas of computer-aided design, artificial intelligence, operating system, database management system, and computer network.

Dr. Kawato is a member of the IEEE, the ACM, Information Processing Society of Japan, and the Institute of Electronics, Information and Communication Engineers.