Evaluating Object-Oriented Query Languages

DANIEL K.C. CHAN AND PHILIP W. TRINDER AND RAYMOND C. WELLAND

Computing Science Department, Glasgow University, Glasgow G12 8QQ, United Kingdom

Different query languages have been implemented and others proposed for object-oriented database systems. Evaluating and comparing these languages has been difficult due to the lack of a frame of reference. This paper establishes such a framework using four dimensions: support of object-orientation, expressive power, support of collections, and usability. Each dimension is defined in terms of a number of criteria. The criteria are, in turn, explained using example queries written in a concise, expressive, and clear query notation: object comprehensions. These same examples also demonstrate the process of evaluating a query language by showing how the criteria can be assessed. An evaluation based on the proposed framework reveals that many well-known query languages do not meet all the criteria. The evaluation framework can also be used constructively in improving existing query languages and directing new query language design.

Received March 1994; revised October 1994.

1. INTRODUCTION

Relational completeness was proposed in [24] and since then it has served as the yardstick for evaluating the expressive power of relational query languages. Later studies of the generalisation of the relational model extended the relational algebra with extra operations. For instance, *replace*, *set-collapse*, and *powerset*, were introduced in [1] to characterise the expressive power of query languages for nested-relational and complex-object models, such as $\neg$1NF [58], NF^2 [56], and VERSO [59]. With the advent of newer data models supporting richer constructs, new definitions of completeness are constantly sought for.

So far, no definition of completeness has been proposed for object-oriented data models like those found in Iris [46], PROBE [48], GemStone [60], ORION [41], ONTOS [51], ENCORE [72], and O$_2$ [8]). Worse still, the situation is unlikely to change for some time. Attempts have been made to tackle the problem by using different notions of completeness which are data model independent [2], but interpreting them in terms of query language operations is not at all straight forward. Not having a formal definition of completeness makes it difficult to evaluate and compare object-oriented query languages objectively.

Many opinions have been expressed about the central and fundamental issues of object-oriented query languages [7, 40, 42, 11]. Relevant inputs can also be found from sources taking a slightly different standpoint. For example, an evaluation framework for query algebras is proposed in [70] and many of the criteria are applicable to high-level query languages. Language facilities for multimedia data are studied in [47]. Almost all the requirements identified there are equally valid for object-oriented query languages in general. It is encouraging that there is a consensus of opinions from these sources.

However, this pool of ideas has several limitations. Firstly, collection operations like operations on sets or lists are not sufficiently characterised nor is the usability aspect of high-level query languages. Secondly, the requirements are stated in rather esoteric terms hence are open to misunderstanding. Thirdly, some of the requirements cannot be evaluated objectively.

The aim of introducing a new set of criteria is to provide a direct and easily measurable framework as well as to make up for overlooked issues. The guideline adopted is to include only features that can be expressed at the language level. In other words all the criteria can be, and indeed will be, explained using query examples. The virtue of using examples is two-fold. Not only does it depict the evaluation framework, but it also provides a vivid account of evaluating a query language by showing how the criteria of the framework can be assessed.

The examples are expressed in a high-level query language because it is believed that such a language is more comprehensible than for example a formal query language similar to the relational calculus. It is also believed that a new query notation will give a better insight into the needs of object-oriented query languages and avoid the confusion caused by redefining existing query languages. This belief, together with the inadequacies of existing languages, motivates the introduction of a new query notation. As a result, *object comprehensions* are introduced in this paper and used for the example queries.

The structure of the paper is as follows. Section 2 defines an example database that will be used to describe the evaluation framework. Section 3 introduces object comprehensions. Section 4 describes the evaluation framework. Section 5 discusses related issues that are not included in the framework. Section 6 il-

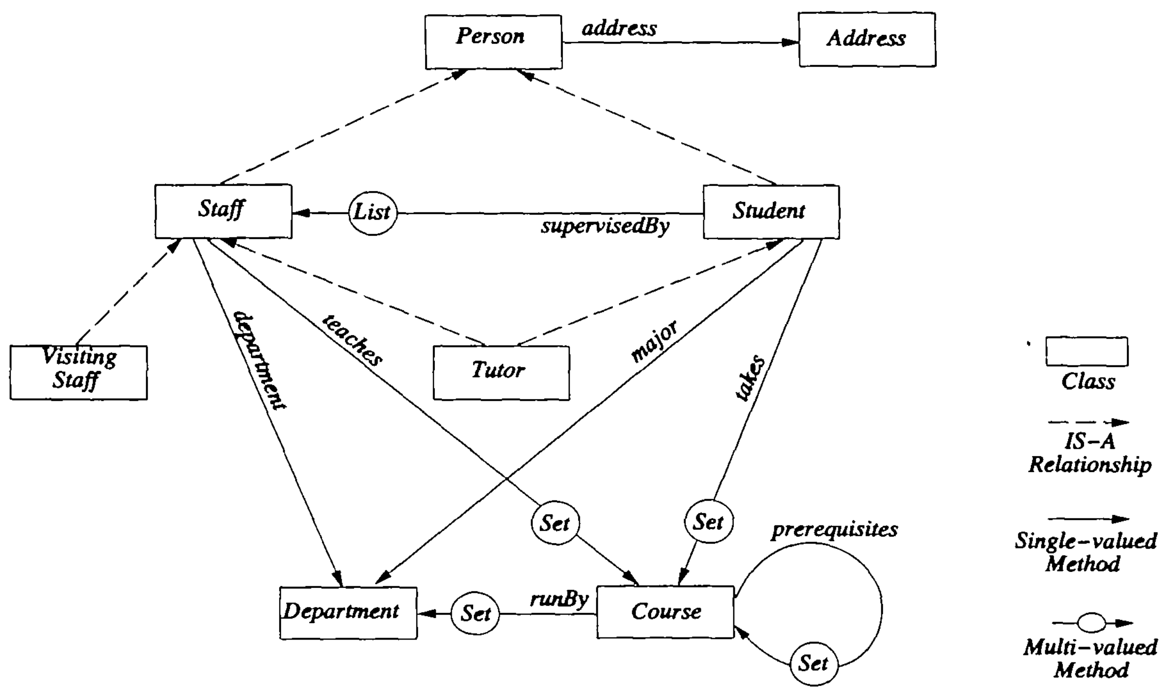

FIGURE 1. Simplified Schema Diagram.

lustrates how a query language can be evaluated using a set of example queries expressed in object comprehensions. Section 7 summarises the evaluation of four well-known object-oriented query languages using the proposed framework. Section 8 concludes.

2. THE RUNNING EXAMPLE

2.1. Reference Data Model

A reference data model whose formal specification can be found in [22] is used to define the example database which will be used throughout the paper. The model fulfills the requirements for object-oriented data models as stated in [6, 7, 28] and therefore is similar to many existing data models [71]. Some features are however worth mentioning: (1) tuples are not supported; (2) class extents are not supported; and (3) three collection classes are supported. One of the reasons of this choice of features is to establish an evaluation framework that can be meaningfully applied to *any* object-oriented query language. The similarities between existing data models are therefore captured and their differences excluded. Support of tuples and class extents significantly simplifies querying; however, they are not supported in some data models. Assuming their existence will limit the applicability of the evaluation framework. On the other hand, it is felt that features not yet well studied should be included to widen the scope of the evaluation framework. Support of multiple collection classes complicates, instead of simplifies, querying. Its inclusion therefore widens, instead of limits, the applicability of the framework. In other words, even though a reference model is used here to define the example database the generality is not lost, so the framework is equally applicable to many other models.

2.2. Example Database

The example database is a simplified university administration system that records information about students and staff members of a university, its academic departments and courses. The relationships between classes defined in the schema are shown in Fig. 1.

The class *Person* has two subclasses: *Student* and *Staff*. *VisitingStaff* is a subclass of *Staff*. *Tutor* inherits from both *Student* and *Staff* to represent students doing part-time teaching. Every person is given an address which is an object of class *Address*. A student can have a principal supervisor, a second supervisor, and so forth. This relationship is modelled by the method *supervisedBy* as a list of staff members. Every staff member and student is associated to an academic department of class *Department* via *department* and *major* respectively. Courses given by each staff member and taken by each student are also recorded. They are represented by set-valued methods, *teaches* and *takes*. A course may have a set of prerequisite courses (*prerequisites*) and is administered by one or more academic departments (*runBy*). A course is an instance of the class *Course*.

The schema definition is given in Fig. 2. In order to keep it simple, only the relevant method signatures are given, attributes and method implementations are omitted. The reference data model supports a root class, *Entity*, which is a direct or indirect superclass of all other classes. The calculation of the salary of a tutor is different from that of a staff member. This variation is captured by giving an overloaded method *salary* to *Tutor*. Also recorded is the number of credits (*credits*) each course is worth and the percentage weights of assessments given in each course. The latter is represented as a bag of integers using the method *assessments*. It is assumed that the database contains six set collections: *Persons*, *Departments*, and *Courses*, containing all instances of their corresponding classes that are members of the university; and *StaffMembers*, *Students*, and *Tutors*, containing instances of the corresponding classes that are members in the Science Faculty.

3. OBJECT COMPREHENSIONS

Object comprehensions are developed from *list comprehensions*, which are constructs based on the standard mathematical notation for sets. List comprehensions are widely available in functional programming languages, e.g. [66, 67, 36] and have been found very useful in manipulating collections of data. A full description can be found in [55]. They have also been applied to imperative languages such as an experimental version of PS-algol [65]. Recent research on functional databases [63, 57, 62] uses comprehensions as a query facility and research on semantic data models [54, 32] uses comprehensions to support query processing.

It has been argued convincingly in [64] that comprehensions are a good query notation, being concise, expressive, clear, optimisable, and well integrated with programming languages. A study of bulk types in [69] shows that comprehensions can provide a uniform query notation over many suitably defined collections. An extension to incorporate local definitions in comprehensions is recommended in [33]. Side-effecting qualifiers that permit data to be manipulated by side-effects in addition to being queried are proposed in [31]. More importantly side-effecting qualifiers also allow "local" optimisation.

Comprehensions are very similar to SQL, on which many object-oriented query languages are based. In fact, the idea that comprehensions are a formal and clean notation for SQL was first floated in [10].

The object comprehensions presented in this paper are an extension to support querying object-oriented databases. More details, including syntax, semantics, optimisation, translation from other query languages, and algebraic support, can be found in [23, 17].

Using the running example, a query returning the course code of courses having more than two credits can be expressed as follows,

Q0. Return the course codes of courses having more than 2 credits.

Set[*c* ← *Courses; c.credits* > *2* | *c.code*]

```
Class Person isa Entity
methods
name:  → String,
address: → Address.

Class Staff isa Person
methods
department:  → Department,
teaches:  → Set of Course,
salary:  → Integer.

Class Student isa Person
methods
major:  → Department,
supervisedBy:  → List of Staff,
takes:  → Set of Course.

Class Tutor isa Staff, Student
methods
salary:  → Integer.

Class VistingStaff isa Staff.

Class Department isa Entity
methods
name:  → String.

Class Course isa Entity
methods
code:  → String,
runBy: → Set of Department,
prerequisites:  → Set of Course,
assessments:  → Bag of Integer,
credits: → Integer.

Class Address isa Entity
methods
street:  → String,
area: → String,
city: → String.

Database is
Persons: Set of Person,
Departments:  Set of Department,
Courses:  Set of Course,
StaffMembers:  Set of Staff,
Students:  Set of Student,
Tutors:  Set of Tutor.
```

FIGURE 2. Simplified Schema Definition.

The result of evaluating the above comprehension query is a new collection, precisely a set, computed from the existing collection *Courses* of class *Set of Course*. The elements of the new collection are determined by repeatedly evaluating *c.code*, as controlled by the qualifier *c.credits* > *2*. Since the result of *c.code* is of class *String* the elements in the resultant set are therefore of class *String*. Comprehension syntax can be sketched as follows: a comprehension starts with a collection class, e.g. set, followed by '[' and a sequence of qualifiers.

A qualifier is either a *filter*, *generator*, or *local definition*. A filter is just a boolean-valued expression imposing a condition that must be satisfied for an element to be included in the result. An example of a filter was *c.credits* > *2* above, ensuing that only courses having more than two credits are used in computing the result. A generator of the form $V \leftarrow E$, where E is a collection-valued expression, makes the variable V range over the elements of the collection. An example of a generator was $c \leftarrow$ *Courses* above, making c range over the elements of the set *Courses*. A local definition of the form *N as E*, introduces a symbolic name N for the value of the expression E. An example of the use of this construct can be found in query Q29 in Section 6.

4. THE EVALUATION FRAMEWORK

The evaluation framework uses four dimensions, namely *support of object-orientation*, *expressive power*, *support of collections*, and *usability*. Each dimension is defined in terms of a number of criteria.

Support of object-orientation measures the support given to the intrinsic properties of object-oriented data models. There is an almost unanimous agreement in the publication about features under this category. They include

- Object Identity
- Method Calling
- Complex Objects
- Class Hierarchy
- Dynamic Binding

In the object-oriented paradigm, an object is identified by an unique and immutable object identifier which is independent of the "contents" of the object. To support objects, a query language needs to operate on object identifiers, for example, the equality over object identifiers.

Objects are encapsulated meaning that their "contents" cannot be accessed directly and all accesses must be done via methods defined for the objects. The association of specific methods to objects is a fundamental tenet of the paradigm, the use of methods in a query should therefore be supported.

In contrast to the simple attributes of a tuple in the relational model, an object can be perceived as a complex entity. Applying a method on such a complex object can result in the return of a base value, an object, or a collection. A query language supporting method calling should therefore also accommodate results of different types.

The class hierarchy defines a classification scheme based on specialisation over classes. This naturally leads to the adoption of the substitutional semantics which conceals the differences of objects originated from different classes along the same specialisation chain. On the other hand, the class hierarchy contains useful information regarding the classification of objects which may form the basis of a query. A query language should therefore provide a mechanism with which the classification information can be exploited.

The class hierarchy also introduces the notion of inheritance where methods defined in a superclass are inherited by all its subclasses. Moreover, method overloading allows different methods to be given the same name. Given the substitutional semantics and the possibility of method overloading, the selection of a method can only be determined dynamically. A query language should therefore support dynamic binding of methods or behave as if methods were dynamically bound.

Expressive power examines the ability to explore and synthesize complex objects and collections. Attention is mainly drawn to the manipulation of individual objects. There is quite a reasonable consensus in the literature regarding features in this category which are listed below,

- Multiple Generators
- Dependent Generators
- Returning New Objects
- Nested Queries
- Quantifiers
- Relational Completeness
- Nested Relational Extension
- Recursion

A query often involves one or more collections. The ability to specify more than one *domain* collection - using multiple generators - in a query is as natural and important as in earlier data models. Not supporting multiple generators will result in a more procedural query language relying heavily on query nesting and query functions, if they are supported. Consequently queries are more difficult to express. One language taking this approach is LIFOO [14].

A collection can be returned as the result of a method call. To query such a "nested" collection, a query language should be able to express dependency between generators. Generally speaking, in the absence of dependent generators, the fact that an object is an element of a nested collection has to be "re-established" resulting in more verbose queries, see OSQL [45] for example.

So far, there has been no convincing argument from the modelling perspective about restricting a query language to return only existing objects. Here a query language is required to allow new objects to be created in a query. It is not required that the corresponding classes are created along with the objects. In other words, closure at the instance level should be respected while closure at the class level is not required. To be more precise, no operations for the creation of new classes or the manipulation of the class hierarchy are required. This decision is heavily influenced by the fact that dynamic class creation is still an outstanding problem with no satisfactory solution. Detailed discussion of this controversial issue is given in [20] where the support of views is discussed.

Nested queries are crucial in the construction of new objects especially complex objects. It has been shown that many nested queries in SQL which appear only in the *where* clause (the filter) can be eliminated [39, 30]. In an object-oriented query language supporting free nesting of queries, it it not obvious how nested queries can be eliminated without other language constructs such as query functions. Query nesting can also be considered as an issue of generality.

Quantifiers can simplify queries and provide optimisation opportunities. Quantifiers can be simulated in many query languages; however, their optimisation always involves matching of large patterns as in [43]. Quantifiers can significantly simplify the manipulation of different kinds of collections.

Object-oriented data models subsume the relational model, an object-oriented query language should similarly subsume relational completeness. One possible definition of this requirement is that if the data are relations, a query language should be able to express all queries that can be expressed in the relational algebra. However, a more general definition would be more appropriate for comparing query languages for object-oriented databases. Basically, the concept of a relation being a set of tuples can be replaced by a collection of objects. A query language should therefore be able to express whatever can be expressed in the relational algebra for collections and objects. Inevitably, the relational algebra operations will be more restrictive if tuples are not supported. They should also behave slightly differently for different collection kinds.

Studies of the generalisation of the relational model result in the introduction of three extra operations: *replace*, *set-collapse* and *powerset*. It has been shown that *powerset* incurs superexponential complexity [37] which justifies its omission from the requirement list. The other two operations should be supported and their definitions can be similarly generalised as the relational algebra operations.

In the object-oriented paradigm, cyclic relationships can be defined by one or more methods. Some form of recursion, for instance, transitive closure, should be supported to enable cyclic relationships to be explored.

Support of collections looks into the features that are required for queries involving multiple collection classes. The quest is to find a set of good generic operations that behave consistently for different collection classes. Equally important is the mixing of and conversion between different collection classes. Support of collections looks into the following features

- Collection Literals
- Collection Equality
- Aggregate Functions
- Positioning & Ordering
- Occurrences & Counting
- Converting Collections
- Combining Collections
- Mixing Collections

In some data models including the reference data model, collections are represented as objects hence their comparison is based on object identifiers. However, collections are very often characterised by their contents and behave like base values. This suggests that collections should be allowed to have dual behaviour. One aspect of this duality is to allow collection literals to be expressed. Collection literals can be simulated in some query languages; however, providing direct support simplifies queries as has been shown in SQL [27]. Using the same argument, it should be possible to compare two collections based on their elements instead of their identifiers. Aggregate functions return a value from a collection and have been shown very useful in earlier data models. When ordered collections are supported, a query language should be able to express queries related to a position in the order and the ordering between two elements. When collections are allowed to have duplicates, a query language should be able to return objects with a particular number of occurrences and to count the number of occurrences of an object. It is also important to allow collections to be combined, converted, and mixed within a query.

Usability focuses on the ease of use of a query notation which is essential to the success of a high-level query language. The criteria in this category are

- Local Definitions
- Query Functions

Long path expressions are not uncommon in object-oriented query expressions. To avoid repeating long path expressions, "shorthands" can be introduced using local definitions. Complicated queries are easier to express in an incremental fashion. Query functions allow a complicated query to be broken down into smaller and more comprehensible subqueries.

5. RELATED ISSUES

The previous discussion focuses on query language features without addressing the impact of data model on query languages. There has been much discussion about the advantages and disadvantages of supporting class extents [40, 3]. Class extents create security problem as all instances of a class can be accessed via the class extent and access control on individual objects is difficult and prohibitively expensive. Application modelling often does not require the use of class extents. The provision of class extents has a great impact on what a query language can retrieve. Querying a database becomes easier as every class extent provides an entry point to the database and every object is guaranteed to be directly accessible from at least one class extent. The result is a simpler query language and more optimisation opportunities. Some object-oriented data models, including the reference data model, do not support class extents. To have a set of criteria that are generally applicable, the existence of class extents cannot be assumed. The criteria given in the previous section are derived with no assumption of class extents. For data models supporting class extents, some of the requirements will become superfluous. For example, using multiple generators together with the membership test on collections, dependent generators can be simulated and the class hierarchy can be supported.

Different kinds of equality have been introduced for testing the equivalence of objects based on their contents so that optimisation can be done with more flexibility. The deep-sensitive equivalence rules described in [61, 50] are very complicated and significantly increase the search space of the optimiser. The essence is that these different kinds of equality may be useful in query algebras but their necessity in high-level query languages is questionable.

Many other features of object-oriented query languages have been suggested [26, 13, 7, 70, 47, 68] and are enumerated below. They are useful guidelines for the design of query languages. However, using them as requirements for evaluating query languages is less effective and their assessment can be difficult.

- Simple
- Elegant
- Closed
- Application Independent
- Well Integrated
- Formal Semantics
- Optimisable
- Extended Facilities
- Data Administration
- Integrity Constraints
- Computationally Complete
- Consistent
- General
- Adequate
- Orthogonal
- Strongly Typed
- Efficient
- Null Values
- Versions
- Schema Evolution
- Rules & Triggers

A consistent notation encourages similar concepts or problems to be expressed in similar ways. A query language is general if it allows free composition of constructs and does not impose arbitrary restrictions. Closure refers to the fact that the result of a query can be similarly manipulated by the query language. In a mono-type model, like the relational model, it is a necessary and sufficient requirement. When multiple types are supported, as in object-oriented data models, closure becomes a necessary but not sufficient requirement. Adequacy provides the sufficiency by requiring a query language to operate on all types supported in the data model. Orthogonality is usually used in discussing persistence meaning that a query language can work on both persistent as well as transient data.

6. EVALUATING QUERY LANGUAGES

The four dimensions used in the evaluation framework are illustrated in the next four subsections. The criteria of each dimension are demonstrated using example queries. The point being illustrated by each query is underlined. An explanation is given after each query. Queries involving students, staff members, and tutors should be read as students, staff members, and tutors of the Science Faculty, unless stated otherwise.

6.1. Support of Object-Orientation

6.1.1. Method Calling

Q1. Return staff members named Steve Johnson.

$Set[\ s \leftarrow StaffMembers;$
$\quad \underline{s.name} = \textit{“Steve Johnson”} \mid s\ ]$

Encapsulation protects attributes of an object from being accessed directly. Such an access must be made via a method. In Q1, *s.name* represents the calling of method *name* on a staff member object *s* drawn from the collection *StaffMembers*.

6.1.2. Dynamic Binding

Q2. Return staff members earning more than £2000 per month.

$Set[\ s \leftarrow StaffMembers;\ s.\underline{salary} > 2000 \mid s\ ]$

Recall that a tutor is a staff member whose salary is calculated differently using an overloaded method. Since *StaffMembers* may contain tutor objects, the calling of method *salary* on an object drawn from it may result in different methods being dynamically bound at run-time.

6.1.3. Complex Objects

Q3. Return tutors living in Glasgow.

$Set[\ t \leftarrow Tutors;\ \underline{t.address.city} = \textit{“Glasgow”} \mid t\ ]$

Support of complex objects implies that a method call may return an object. The returned object can, in turn, receive another method call. This can go on for several method calls until, for instance, a base value is returned. In Q3, *t.address.city* represents the calling of method *city* on the result returned by calling *address* on a tutor object *t*.

6.1.4. Object Identity

Q4. Return tutors working and studying in the same department.

Set[$t \leftarrow$ *Tutors*; *t.department* $\equiv$ *t.major* | *t*]

In object-oriented data models, objects are represented by object identifiers which are essential for object sharing and representing cyclic relationships. Equality between objects becomes equality between object identifiers. In Q4, the equality operator, "=", is overloaded to compare two department objects using their object identifiers.

6.1.5. Class Hierarchy

Q5. Return all visiting staff in the university.

Set[$p \leftarrow$ *Persons*; *p* *hasClass* *VisitingStaff* | *p*]

There is no collection in the database containing only objects of class *VisitingStaff*. *StaffMembers* contains only members in the Science Faculty. The only collection that contains all visiting staff members is *Persons*. It is the reason why the *Persons* collection is used in this query. Since a collection can contain heterogeneous elements belonging to different classes, elements of *Persons* can be of class *Person* or its subclasses. One way of selecting elements from such a collection is to specify the class of interest. In Q5, *hasClass* returns true if person object *p* is indeed of class *VisitingStaff*. This operation is essential for data models not supporting class extents.

Q6. Return all visiting staff members in the university who earn more than £2000 per month.

Set[$p \leftarrow$ *Persons*;
    p *hasClass* *VisitingStaff* *with* *p.salary* > *2000*
    | *p*]

The method *salary* is defined for visiting staff members but not persons in general. Therefore calling *salary* on a person object may result in an error. To allow selection that is applicable only to objects of a particular class, the *hasClass & with* construct can be used. The role of *with* is similar to that of conjunction. The second condition (e.g. *p.salary* > *2000*) is evaluated only if the first condition (e.g. *p hasClass VisitingStaff*) is true. It the first condition is false the whole construct returns false. However, the conditions around *with* cannot be swapped. In other words, *with* is a non-symmetric conjunction. This construct is essential for supporting static type checking in the absence of support for class extents.

6.2. Expressive Power

6.2.1. Multiple Generators

Q7. Return students studying in the same department as Steve Johnson.

Set[$x \leftarrow$ *Students*; $y \leftarrow$ *Students*;
    x.name = *"Steve Johnson"*;
    x.major = *y.major* | *y*]

Multiple generators allow relationships that are not explicitly defined in the database schema to be "re-established". In Q7, x is ranged over *Students* and y is ranged over the same set but independently. The missing relationship is established using the major departments of x and y. By and large, not supporting multiple generators results in a language having a stronger procedural favour and is often more difficult to use. LIFOO [14, 15], which for the most part, does not support multiple generators, is a good example [18]. More importantly, multiple generators are essential for constructing new objects from existing ones.

6.2.2. Dependent Generators

Q8. Return courses taken by the students.

Set[$s \leftarrow$ *Students*; $c \leftarrow$ *s.takes* | *c*]

The result of a method call can be a collection object containing many elements. To facilitate querying over the elements in such a "nested" collection - which depends on the current binding in the first collection - a dependent generator can be used. In Q8, c ranges over the collection returned by calling *takes* on the current student object s (i.e. the element in *Students* that is currently bound to s).

6.2.3. Returning New Objects

Q9. Return students and the courses taken by them. The result is obtained by creating new objects using the student objects and the sets of courses.

Set[$s \leftarrow$ *Students* | *AClass.new(s, s.takes)*]

So far, only queries returning existing objects have been examined. To return "new" information, the corresponding class has to be defined *beforehand* and the query will create objects of this class as the result. In

Q9, the method *new*, which takes two parameters: *s* and *s.takes*, is called on the class *AClass*.

Many systems, e.g. IRIS, would use *tuples* to return the result of Q9. The drawbacks with this approach are (1) tuples are values and duplicates are always eliminated and (2) new objects have to be generated using perhaps a programmatic interface. This is unsatisfactory in terms of semantics and functionality. Another approach that solves a subset of the general problem involves creating new classes and factorising existing classes [49, 34, 4]. Detailed discussion on this issue can be found in [35, 68, 20].

6.2.4. *Nested Queries*

Q10. Return students and the courses taken by them that have more than one credit. The result is obtained by creating new objects using the student objects and the sets of courses.

```
Set[ s ← Students | AClass.new( s,
         Set[ c ← s.takes; c.credits > 1 | c ] ) ]
```

Nested queries enable richer data structures to be returned as well as complex selection conditions to be expressed. In Q10, the inner query returns a set of courses and is used as a parameter to the method call in the result expression of the outer query.

6.2.5. *Quantifiers*

In order to provide a coherent notation for querying over different collection classes, object comprehensions rely on quantifiers to express many collection operations. The quantifiers introduced here concerns the occurrences of collection elements and they have the same semantics for sets, bags, and lists.

Q11. Return students taking a course given by Steve Johnson.

```
Set[ l ← StaffMembers;
     l.name = "Steve Johnson"; s ← Students;
     some s.takes = some l.teaches | s ]
```

The quantifiers in object comprehensions can be seen as operator modifiers providing a family of new operators. In Q11, the last filter succeeds if there is a common course element between the sets: *s.takes* and *l.teaches* (i.e. an non-empty intersection)*.

Q12. Return students taking only courses given by Steve Johnson.

```
Set[ l ← StaffMembers;
     l.name = "Steve Johnson"; s ← Students;
     every s.takes = some l.teaches | s ]
```

In Q12, the last filter is true if all the course elements in *s.takes* are also in the set *l.teaches*†. Q12 expresses the subset relation.

6.2.6. *Relational Completeness*

Object-oriented data models provide rich data constructs that can subsume those found in the relational model. Operators that make a language relationally complete are equally useful and important in the object-oriented setting. The five primitive operations: *selection*, *projection*, *cross product*, *union*, and *difference*, which make a language relationally complete can be expressed in their equivalent forms in object comprehensions. *Selection* has been demonstrated in many queries, e.g. Q1, so only the other four operations are given below,

Q13. Return the names of students.

```
Set[ s ← Students | s.name ]
```

Q14. Return all the possible combinations between departments and courses.

```
Set[ d ← Departments; c ← Courses
     | AClass.new( d, c ) ]
```

Q15. Return staff members and students in the Computing Science Department.

```
Set[ s ← StaffMembers;
     s.department.name = "Computing" | s ]
union
Set[ s ← Students;
     s.major.name = "Computing" | s ]
```

Q16. Return areas where students, but no staff, live.

```
Set[ s ← Students | s.address.area ]
differ
Set[ s ← StaffMembers | s.address.area ]
```

Projection is demonstrated in Q13 where *s.name* "projects" the name of a student object. *Cross product* between *Departments* and *Courses* is expressed in Q14 using two generators and by returning new objects. With object comprehensions projection is limited to single objects. Otherwise, new objects have to be constructed using the projected objects as in Q14. A new collection is constructed from two collections in Q15 using *union*. The difference between two collections can be expressed using *differ* as in Q16. For *union* and *differ* the elements of the resultant collection will be of the most specific unique common superclass of the element classes in the operand collections.

*Semantics: $\exists x \exists y \bullet x \in s.takes \wedge y \in l.teaches \wedge x = y$.

†Semantics: $\forall x \exists y \bullet x \in s.takes \wedge y \in l.teaches \wedge x = y$.

6.2.7. *Nested Relational Extension*

Studies of the generalisation of the relational model result in the introduction of three extra operations: *replace*, *set-collapse* and *powerset*. It has been observed that *powerset* is only used in constructing domains for the proofs and is not required in the actually manipulation of data. It is therefore chosen not to make it a required feature. *Set-collapse*, which turns a set of sets into a set containing all elements in the original set elements, has been shown in Q8. *Replace*, which applies a function to every element of the input collection, is demonstrated below,

Q17. Return income tax of staff as 40% of their salaries.

```
let tax( w : Integer ) be
    w × 0.4
in Set[ s ← StaffMembers | tax( s.salary ) ]
```

Strictly speaking it is not necessary to express the query in this way. The use of a query function is to emphasise the point that a function is being applied to the salary of the staff members.

6.2.8. *Recursion*

Q18. Return all direct and indirect prerequisite courses for the "DB4" course.

```
let f( cs : Set of Course ) be
    cs
    union
    Set[ x ← cs; y ← f( x.prerequisites ) | y ]
in Set[ c ← Courses; c.code = "DB4";
        p ← f( c.prerequisites ) | p ]
```

In object-oriented data models, it is possible to find cyclic relationships between one or more classes. This suggests that recursive queries should be supported. With object comprehensions, recursive queries can be expressed using query functions. In Q18, the result of the query is generated by retrieving elements, *p*, from a collection returned by a recursive function, *f(c.prerequisites)*. Function *f* takes a set of courses and returns a set of courses. For each element *x* drawn from the input collection *cs*, *f* is applied recursively on the prerequisite courses of *x*, *x.prerequisites*, and the result is then used as part of the input. The recursion stops when the input collection to *f*, *cs*, is an empty set.

6.3. Support of Collections

6.3.1. *Collection Literals*

Q19. Return students living in the following areas: Hillhead, Kelvinside and Dowanhill.

```
Set[ s ← Students; s.address.area = some
    Set{ "Hillhead", "Kelvinside", "Dowanhill" }
    | s ]
```

Collection literals can simplify queries by making them more concise and arguably clearer. In Q19, a set literal of strings is specified by listing the elements within curly brackets. A collection literal can be seen as a shorthand for the union of a number of constant object comprehensions.

6.3.2. *Collection Equality*

Q20. Return courses with no prerequisite courses.

```
Set[ c ← Courses; c.prerequisites == Set{} | c ]
```

In many occasions it is necessary to compare two collections based on the elements, their occurrences, and their order. Two bags are equal if for each element drawn from either collection there is equal number of occurrences in both bags. For lists, the number of occurrences and the positions must be the same. In Q20, the filter returns true if *c.prerequisites* is an empty set. Note that object comprehensions do not support equality on objects that are not collections.

6.3.3. *Aggregate Functions*

Q21. Return courses with less than two assessments.

```
Set[ c ← Courses; (size c.assessments) < 2 | c ]
```

The aggregate function *size* returns the number of elements in a collection. It is defined for all collection classes. For bags and lists duplicate elements are included in the counting. Some aggregate functions are however defined only for some collection classes.

6.3.4. *Positioning and Ordering*

A list allows duplicate elements and keeps track of the order of the elements. Naturally queries involving lists may question on the order or positions of elements.

Q22. Return the first and second supervisors of Steve Johnson.

```
Set[ s ← Students; s.name = "Steve Johnson";
    i ← List{ 1..2 } | s.supervisedBy.[ i ] ]
```

In Q22, the first two elements of a list are returned using a collection literal as a generator.

Q23. Return students having Steve Johnson before Bob Campbell in their supervisor lists.

```
Set[ s ← Students;
     i ← List{ 1..(size s.supervisedBy) };
     s.supervisedBy.[i].name = "Steve Johnson";
     j ← List{ 1..(size s.supervisedBy) };
     s.supervisedBy.[j].name = "Bob Campbell";
     i < j | s ]
```

In Q23, a list literal is generated using the size of the list returned by *s.supervisedBy*. Two generators are ranged over it to match the given names. The relative order is determined using the range variable *i* and *j*.

6.3.5. *Occurrences and Counting*

Bags and lists allow duplicate of elements. Q24 and Q25 show how the occurrences of elements can be used and retrieved using object comprehensions.

Q24. Return courses with 4 assessments of the same percentage weight.

```
Set[ c ← Courses;
     i ← c.assessments;
     just 4 c.assessments = i | c ]
```

In Q24, the selection is based on the number of occurrences (i.e. *4*) of an element (i.e. *i*) in the collection *c.assessments*. The use of numerical quantifiers simplifies retrieval based on occurrences.

Q25. Return the number of assessments worth 25% in the course DB4.

```
Set[ c ← Courses; c.code = "DB4";
     i ← List{ 1..(size c.assessments) };
     just i c.assessments = 25 | i ]
```

In Q25, the number of occurrences (i.e. *i*) of a given element (i.e. *25*) in the collection *c.assessments* is returned. The possible number of occurrences are generated using a literal generator ranging from 1 to (*size c.assessments*).

6.3.6. *Converting Collections*

Q26. Return the salary of tutors and keep the possible duplicate values.

```
Bag[ t ← Tutors | t.salary ]
```

This query is based on a set of tutor objects and therefore the result is naturally a set of integers containing no duplicate values. If duplicates are to be kept the result can be specified to be a *bag*. Explicitly specifying the resultant collection kind provides a high-level mechanism to manage duplicates. Otherwise, implicit conversion rules have to be imposed or explicit conversion of all generators to the resultant collection kind will be required. Converting a collection into a set results in the elimination of duplicates and the loss of the order between elements. Converting a collection into a bag keeps the number of elements unchanged - duplicates are not lost and no new elements are introduced - but the order between the elements is lost. Converting a collection into a list keeps the number of elements and an arbitrary order is assigned to the elements.

6.3.7. *Combining Collections*

Q27. Return the students supervised by Steve Johnson. The result should include only those with Steve Johnson as either the first or second supervisor. The ones having him as the first supervisor should be listed before the ones having him as the second supervisor.

```
List[ l ← StaffMembers; l.name = "Steve Johnson";
      s ← Students; s.supervisedBy.[1] = l | s ]
union
List[ l ← StaffMembers; l.name = "Steve Johnson";
      s ← Students; s.supervisedBy.[2] = l | s ]
```

It has been shown that *union* can be used to combine two sets to return a set containing all the elements of the original sets. Here *union* is used to combine two lists. It appends the second list to the first one.

6.3.8. *Mixing Collections*

Q28. Return courses taught by the supervisors of Steve Johnson.

```
Set[ s ← Students; s.name = "Steve Johnson";
     sup ← s.supervisedBy; c ← sup.teaches | c ]
```

If an object-oriented data model supports more than one collection kind, the corresponding query notation should support not only different collection kinds but also the mixing of them in the same query. In Q28, the first generator is drawn from the set *Students*, the second generator from the list *s.supervisedBy*, and the last generator from the set *sup.teaches*. Knowing the resultant collection kind, object comprehensions can automatically convert all generators into the resultant collection kind.

6.4. Usability

6.4.1. *Local Definitions*

Q29. Return students whose major departments are in either Hillhead Street or University Avenue.

```
Set[ s ← Students; a as s.major.address.street;
     a = "Hillhead Street" or
     a = "University Avenue" | s ]
```

Local definitions simplify queries by providing symbolic names to expressions. They are particularly useful when an expression is used in more than one place. In Q29, *s.major.address.street* would have been written twice if local definitions were not supported. The use of the symbolic name *a* for the expression saves repeating the long expression twice.

6.4.2. Query Functions

Q30. Return students taking some course run by their departments.

let cs(d : Department) be
Set[c ← Courses; some c.runBy = d | c]
in Set[s ← Students;
some s.takes = some cs(s.major) | s]

Query functions are crucial for expressing recursive queries as shown in Q18. They also help in organising a query; the resultant query is often more comprehensible. In Q30, a nested query is separated into two simple queries and one of them is expressed as a query function.

7. AN EVALUATION

In this section a summary of an evaluation of four well-known query languages using the proposed framework is presented. The languages evaluated are the ONTOS[51, 53, 52], Iris[29, 46, 9], ORION[41], and O_2[5, 8] query languages. This paper is not a commentary on these particular languages and interested readers are referred to the sources listed above. These query languages are chosen as representative languages mainly because they are well-reported and the most referenced in the literature. Readers are very likely to know one of these languages and the evaluation would therefore be more meaningful than using less well-known languages. The result of the evaluation is summarised in the following tables.

TABLE 1. Support of Object-Orientation.

	ONTOS	IRIS	ORION	O_2
Method Calling	√	√	√	√
Dynamic Binding	√	√	√	√
Complex Objects	√	√	√	√
Object Identity	√	√	√	√
Class Hierarchy		√	√	√

Method calling is supported by all the four query languages. OSQL, ORION and O_2SQL also support direct access to attributes. OSQL supports the class hierarchy via class extents and membership test. ORION provides four constructs to support the class hierarchy: (1) class extents and the membership test operation *is-in*; (2) the operations * (meaning including instances of all subclasses), *union*, and *difference* over class extents to form class extent expressions; (3) specifying the class of the object returned by a method call using *class*, this specification can be sandwiched between method calls within the same path expression; and (4) specifying the class of objects used and returned in a recursive query using *is-a*. The ONTOS and O_2 data models support class extents only as an option. ONTOS SQL does not support the class hierarchy properly in its current form. In other words, support given to the class hierarchy partly depends on how the schema is defined. It is however possible to extend ONTOS SQL to support the class hierarchy using the available interfaces for collection classes and the database.

TABLE 2. Expressive Power.

	ONTOS	IRIS	ORION	O_2
Multiple Generators	√	√	√	√
Dependent Generators			√	√
Returning New Objects				
Nested Queries		√		√
Quantifier			√	√
Selection	√	√	√	√
Projection	√	√		√
Cartesian Product	√	√		√
Union				√
Difference				√
Set-collapse		√	√	√
Replace		√		√
Recursion				√

IRIS supports class extents and hence as explained previously the support of dependent generators in OSQL is not strictly necessary. On the other hand, the expressive power of ONTOS SQL suffers badly because class extents are optional and dependent generators are not supported. None of the query languages supports the return of new objects. It is a result of the limitation of current technology since creating new objects is such an expensive operation that can significantly slow down query processing. ONTOS SQL can only return either a string or a list of strings, while OSQL and O_2SQL often use tuples to return new "objects". ORION does not seem to offer a solution in this aspect. Nested queries can only appear in filters of an OSQL query. Generally speaking a nested query in a generator can always be eliminated and hence a nested query is most useful when it is used in filters or the result expression.

Quantifiers can be simulated provided that nested queries are supported in addition to the membership test and the cardinality operations of collections. ONTOS SQL does not support nested queries and hence cannot simulate quantifiers. OSQL can simulate the use of quantifiers in filters. If quantifiers are used elsewhere "foreign" functions - implemented in a programming language - can be employed.

ONTOS SQL does not support *union* and *differ*. ORION does not seem to support *projection* and the proposal [41] did not make it clear. It cannot return new objects or tuples (tuples are not supported by its data model) and therefore cannot express *cartesian product*. O_2SQL provides *differ* for sets but not lists. ONTOS SQL does not support *set-collapse*. For the other languages, *set-collapse* can be performed in various ways including implicit flattening.

The four query languages all support some form of *replace*. Functions can be used in ONTOS SQL, OSQL, and O_2SQL while methods can be used in all of them. ORION supports traversal recursion that involves traversal of a cyclic relationship; however, it does not support computational recursion where computation is done along the traversal of a cyclic relationship.

TABLE 3. Support of Collections.

	ONTOS	IRIS	ORION	O_2
Collection Literals			√	√
Collection Equality	√	√		√
Aggregate Functions				√
Positioning	√	-	-	√
Ordering		-	-	
Occurrences		-	-	
Counting		-	-	
Converting Collections		-	-	√
Combining Collections		-	-	
Mixing Collections	√	-	-	√

ONTOS SQL supports only set literals that can appear only in generators. OSQL does not support lists and ORION supports only sets, therefore some entries in the table are not applicable to them. They are marked by a dash (-) in the table. Aggregate functions can be supported using foreign functions in the case of OSQL. O_2SQL can decide whether one element precedes another element in a list with the help of a set literal containing all the positions of the list. It is possible for OSQL and O_2SQL to return objects given the number of occurrence in a collection though in a rather distorted way. Nevertheless, it is simpler for them to return the number of occurrence of a given object. ONTOS SQL does not support conversion between collection classes. The result of a query is either a string or a list of strings. With OSQL the result of a query is always a bag but duplicates in a bag can be eliminated. For O_2SQL, if all the generators are drawn from the same collection kind the result will be of the same kind; otherwise the result is a bag. The function *magic* can turn a set into a list while the function *listoset* and the keyword *distinct* and *unique* turn a list into a set.

TABLE 4. Usability.

	ONTOS	IRIS	ORION	O_2
Local Definitions			√	
Query Functions		√		√

ONTOS SQL does not support local definitions or query functions. OSQL supports query functions that can appear only in generators and local definitions are not supported. ORION does not support query functions while O_2SQL does not support local definitions.

Example queries of the four query languages evaluated can be found in [18, 19]. Results of the evaluation of other query languages, e.g. EXCESS (EXODUS [16]), CQL++ (ODE [25]), OQL[X] (Zeitgeist [12]), and XSQL [38], can be found in [21]. The new SQL3 proposal [44] also includes many of the features discussed in this framework.

8. CONCLUSION

The objective of this paper is to present an evaluation framework that can be used to assess object-oriented query languages impartially. The evaluation framework uses four dimensions: support of object-orientation, expressive power, support of collections, and usability. Object-orientation examines the support given to the intrinsic properties of object-oriented data models. Expressive power looks into the capacity to explore and synthesise complex objects and collections. Support of collections concerns the support of multiple kinds of collections and the interaction between them. Usability focuses on the ease of use of a query notation. Each dimension is defined in terms of a number of criteria. The placement of criteria should not be seen as definitive as some of them can belong to more than one dimension.

In order to provide a better understanding of the criteria, the framework was presented using query examples expressed in object comprehensions. This query notation extends list comprehensions to support multiple collection classes, quantifiers, collection literals, local definitions, and various sorts of predicates. In other words, object comprehensions have been shown to satisfy all the criteria in the framework. Indeed, [17] shows that object comprehensions are at least as powerful as the four query languages evaluated with respect to the reference data model.

Four well-known query languages were evaluated using the proposed framework. None of the query languages satisfies all the criteria. It is rather disappointing that the criteria in the object-orientation dimension are not met by all the query languages. Expressive power is poor except for O_2SQL. Perhaps query languages will become more expressive and powerful as better query processing techniques are discovered. Not all of the data models support multiple collection classes. The support of multiple collection classes and their interaction is not yet well understood. At the moment the query languages provide reasonable support for them even though not necessarily in a consistent way. Usability seems to receive a low priority and is not well supported.

Existing object-oriented query languages can be improved along the directions suggested by the evaluation framework. It is hoped that new query language design can benefit from this framework where the criteria provide a set of comprehensive design requirements.

REFERENCES

[1] S. Abiteboul and C. Beeri. On the Power of Languages for the Manipulation of Complex Objects. Technical report, INRIA, France, January 1993.

[2] S. Abiteboul, C. Beeri, M. Gyssens, and D. van Gucht. An Introduction to the Completeness of Languages for Complex Objects and Nested Relations. In S. Abiteboul, P.C. Fischer, and H.-J. Schek, editors, *Nested Relations and Complex Objects in Databases*, volume 361 of *Lecture Notes in Computer Science*, pages 117–138. Springer-Verlag, 1989.

[3] S. Abiteboul, P. Buneman, C. Delobel, R. Hull, P. Kanellakis, and V. Vianu. New Hope on Data Models and Types: Report of an NSF-INRIA Workshop. *ACM SIGMOD Record*, 19(4):41–48, December 1990.

[4] R. Agrawal and L.G. DeMichiel. Type Derivation Using the Projection Operation. *Information Systems*, 19(1):55–68, 1994.

[5] Altaïr, France. *The O_2 Query Language User's Manual*, December 1989.

[6] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, pages 40–57. Elsevier, 1989.

[7] F. Bancilhon. Query Languages for Object-Oriented Database Systems: Analysis and a Proposal. In *Proceedings of the GI Conference on Database Systems for Office, Engineering, and Scientific Applications*, pages 1–18. Springer-Verlag, 1989.

[8] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building An Object-Oriented Database System - The Story of O_2*. Morgan Kaufmann, 1992.

[9] D. Beech. A Foundation for Evolution from Relational to Object Databases. In *Proceedings of the International Conference on Extending Database Technology*, volume 303 of *Lecture Notes in Computer Science*, pages 251–270. Springer-Verlag, 1988.

[10] C. Beeri. New Data Models and Languages - the Challenge. In *Proceedings of the Principles of Database Systems*, pages 1–15. ACM Press, 1992.

[11] E. Bertino, M. Nagri, G. Pelagatti, and L. Sbattella. Object-Oriented Query Languages: The Notion and the Issues. *IEEE Transactions on Knowledge and Data Engineering*, 4(3):223–237, June 1992.

[12] J.A. Blakeley, C.W. Thompson, and A.M. Alashqur. OQL[X]: Extending a Programming Language X with a Query Capability. Technical Report 90-07-01, Texas Instruments Incorporated, U.S.A., November 1990.

[13] T. Bloom and S.B. Zdonik. Issues in the Design of Object-Oriented Database Programming Languages. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 441–451. ACM Press, October 1987.

[14] O. Boucelma and J.L. Maitre. Querying Complex-Object Databases: the LIFOO Functional Language. Technical report, Université de Provence, France, 1989.

[15] O. Boucelma and J.L. Maitre. An Extensible Functional Query Language for an Object-Oriented Database System. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, volume 566 of *Lecture Notes in Computer Science*, pages 567–581. Springer-Verlag, 1991.

[16] M.J. Carey, D.J. DeWitt, and S.L. Vandenberg. A Data Model and Query Language for EXODUS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 413–422. ACM Press, 1988.

[17] D.K.C. Chan. *Object-Oriented Query Language Design and Processing*. PhD thesis, University of Glasgow, U.K., 1994.

[18] D.K.C. Chan, D.J. Harper, and P.W. Trinder. Object-Oriented Query Languages: Data Model Issues, Survey, Comparison, and Analysis. Technical Report DB-92-1, University of Glasgow, U.K., November 1992.

[19] D.K.C. Chan, D.J. Harper, and P.W. Trinder. A Case Study of Object-Oriented Query Languages. In *Proceedings of the International Conference on Information Systems and Management of Data*, pages 63–86. Indian National Scientific Documentation Centre (INSDOC), 1993.

[20] D.K.C. Chan and D.A. Kerr. Improving One's Views of Object-Oriented Databases. In *Proceedings of the Colloquium on Object-Orientation in Databases and Software Engineering*. Elsevier, 1994.

[21] D.K.C. Chan and P.W. Trinder. An Evaluation Framework for Object-Oriented Query Languages. Technical Report DB-93-3, University of Glasgow, U.K., April 1993.

[22] D.K.C. Chan and P.W. Trinder. An Object-Oriented Data Model Supporting Multi-methods, Multiple Inheritance, and Static Type Checking: A Specification in Z. In *Proceedings of the 8th Z User Meeting*, Workshops in Computing Series, pages 297–315. Springer-Verlag, 1994.

[23] D.K.C. Chan and P.W. Trinder. Object Comprehensions: A Query Notation for Object-Oriented Databases. In *Proceedings of the British National Conference on Databases*, volume 826 of *Lecture Notes in Computer Science*, pages 55–72. Springer-Verlag, 1994.

[24] E.F. Codd. Relational Completeness of Database Sublanguages. In *Data Base Systems*. Prentice-Hall, 1972.

[25] S. Dar, N.H. Gehani, and H.V. Jagadish. CQL++: A SQL for the ODE Object-Oriented DBMS. In *Proceedings of the International Conference on Extending Database Technology*, volume 580 of *Lecture Notes in Computer Science*, pages 201–216. Springer-Verlag, 1992.

[26] C.J. Date. Some Principles of Good Language Design. *ACM SIGMOD Record*, pages 1–7, January 1984.

[27] C.J. Date. *A Guide to INGRES*. Addison-Wesley, 1987.

[28] K.R. Dittrich. Object-Oriented Database Systems: The Notion and the Issues. In *On Object-Oriented Database Systems*, pages 3–10. Springer-Verlag, 1991.

[29] D.H. Fishman, J. Annevelink, E. Chow, T. Connors, J.W. Davis, W. Hasan, C.G. Hoch, W.Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M.A. Neimat, T. risch, M.C. Shan, and W.K. Wilkinson. Overview of the

Iris DBMS. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 219–250. ACM Press, 1989.

[30] R.A. Ganski and H.K.T. Wong. Optimization of Nested SQL Queries Revisited. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–33, 1987.

[31] G. Ghelli, R. Orsini, A. Pereira Paz, and P.W. Trinder. Design of an Integrated Query and Manipulation Notation for Database Languages. Technical Report FIDE/92/41, University of Glasgow, U.K., 1992.

[32] P.M.D. Gray, K.G. Kulkarni, and N.W. Paton. *Object-Oriented Databases - A Semantic Data Model Approach.* Prentice-Hall, 1992.

[33] K. Hammond. Definitional List Comprehensions. Technical Report 90/R3, University of Glasgow, U.K., January 1990.

[34] A. Heuer and P. Sander. Classifying Object-Oriented Query Results in a Class/Type Lattice. In *Proceedings of the 3rd Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems*, volume 495 of *Lecture Notes in Computer Science*, pages 14–28. Springer-Verlag, 1991.

[35] A. Heuer and M.N. Scholl. Principles of Object-Oriented Query Languages. In *Proceedings of the GI Conference on Database Systems for Office, Engineering, and Scientific Applications*, pages 178–197. Springer-Verlag, 1991.

[36] P. Hudak and P. Wadler. Report on the Functional Programming Language Haskell. Technical Report 89/R5, University of Glasgow, U.K., February 1990.

[37] R. Hull and J. Su. On the Expressive Power of Database Queries with Intermediate Types. *Proceedings of the Principles of Database Systems*, pages 39–51, 1988.

[38] M. Kifer, W. Kim, and Y. Sagiv. Querying Object-Oriented Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 393–402. ACM Press, 1992.

[39] W. Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems*, 7(3):443–469, 1982.

[40] W. Kim. Object-Oriented Databases: Definition and Research Directions,. *IEEE Transactions on Knowledge and Data Engineering*, 4(3):327–341, September 1989.

[41] W. Kim. *Introduction to Object-Oriented Databases.* MIT Press, 1990.

[42] W. Kim. On Unifying Relational and Object-Oriented Database Systems. In *European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 1992.

[43] A. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *Journal of the Association of Computing Machinery*, 29(3):699–717, 1982.

[44] K.G. Kulkarni. Object-Orientation and the SQL-Standard. *Computer Standards & Interfaces*, 15:287–300, 1993.

[45] P. Lyngbaek. OSQL: A Language for Object Databases. Technical Report HPL-DTD-91-4, Hewlett-Packard Company, U.S.A., January 1991.

[46] P. Lyngbaek and W. Kent. A Data Modelling Methodology for the Design and Implementation of Information Systems. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 6–17. IEEE Press, 1986.

[47] F. Manola. Object Data Language Facilities for Multimedia Data Types. Technical Report TR-0169-12-91-165, GTE Laboratories Incorporated, U.S.A., December 1991.

[48] F. Manola and U. Dayal. PDM: An Object-Oriented Data Model. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 18–25. IEEE Press, 1986.

[49] M. Missikoff and M. Scholl. An Algorithm for Insertion into a Lattice: Application to Type Classification. In *Proceedings of the 3rd International Conference on Foundations of Data Organisation and Algorithms*, volume 367 of *Lecture Notes in Computer Science*, pages 64–82. Springer-Verlag, 1989.

[50] G. Mitchell. *Extensible Query Processing in an Object-Oriented Database.* PhD thesis, Brown University, U.S.A., May 1993.

[51] Ontologic Inc., U.S.A. *ONTOS Developer's Guide*, 1991.

[52] Ontologic Inc., U.S.A. *ONTOS Reference Manual*, 1991.

[53] Ontologic Inc., U.S.A. *ONTOS SQL Guide*, 1991.

[54] N.W. Paton and P.M.D. Gray. Optimising and Executing DAPLEX Queries using Prolog. *The Computer Journal*, 33(6):547–555, 1990.

[55] S. Peyton-Jones. *The Implementation of Functional Programming Languages*, chapter 7, pages 127–138. Prentice-Hall, 1987.

[56] P. Pistor and P. Dadam. The Advanced Information Management Prototype. In S. Abiteboul, P.C. Fischer, and H.-J. Schek, editors, *Nested Relations and Complex Objects in Databases*, volume 361 of *Lecture Notes in Computer Science*, pages 3–26. Springer-Verlag, 1989.

[57] A. Poulovassilis. *The Design and Implementation of FDL, a Functional Database Language.* PhD thesis, Birkbeck College, University of London, 1989.

[58] M. Roth, H. Korth, and D. Batory. SQL/NF: A Query Language for ¬NF Relational Databases. *Information Systems*, 12(1):99–114, 1987.

[59] M. Scholl, S. Abiteboul, F. Bancilhon, N. Bidoit, S. Gamerman, D. Plateau, P. Richard, and A. Verroust. VERSO: A Database Machine Based on Nested Relations. In S. Abiteboul, P.C. Fischer, and H.-J. Schek, editors, *Nested Relations and Complex Objects in Databases*, volume 361 of *Lecture Notes in Computer Science*, pages 27–49. Springer-Verlag, 1989.

[60] Servio Logic Development Corporation, U.S.A. *Programming in OPAL, Version 1.3*, 1987.

[61] G.M. Shaw and S.B. Zdonik. Object-Oriented Queries: Equivalence and Optimization. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, pages 264–278. Elsevier, 1989.

[62] C. Small and A. Poulovassilis. An Overview of PFL. In *Proceedings of the International Workshop on Database Programming Languages*, pages 89–103. Morgan Kaufmann, 1991.

[63] P.W. Trinder. *A Functional Database*. D.Phil thesis, Oxford University, December 1989.

[64] P.W. Trinder. Comprehensions: a Query Notation for DBPLs. In *Proceedings of the 3rd International Workshop on Database Programming Languages*, pages 55–70. Morgan Kaufmann, 1991.

[65] P.W. Trinder, D.K.C. Chan, and D.J. Harper. Improving Comprehension Queries in PS-algol. In *Proceedings of the 1990 Glasgow Database Workshop*, pages 103–119, U.K., 1990. University of Glasgow.

[66] D.A. Turner. Recursion Equations as a Programming Language. In Darlington, Henderson, and Turner, editors, *Functional Programming and its Application*. Cambridge University Press, 1981.

[67] D.A. Turner. Miranda: a Non-strict Functional Language with Polymorphic Types. In *Proceedings of the 2nd Conference on Functional Programming Languages and Computer Architectures*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1985.

[68] R. Unland and G. Schlageter. Object-Oriented Database Systems: State of the Art and Research Problems. In K. Jeffery, editor, *Expert Database Systems*, chapter 5, pages 117–222. Academic Press, 1992.

[69] D. Watt and P.W. Trinder. Towards a Theory of Bulk Types. Technical Report FIDE/91/26, University of Glasgow, U.K., July 1991.

[70] L. Yu and S.L. Osborn. An Evaluation Framework for Algebraic Object-Oriented Data Models. In *Proceedings of the IEEE Data Engineering Conference*, pages 670–677. IEEE Press, 1991.

[71] S.B. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, 1990.

[72] S.B. Zdonik and G. Mitchell. ENCORE: An Object-Oriented Database Systems. *IEEE Data Engineering Bulletin*, 14(2):53–57, June 1991.