# DESIGNING OQL: ALLOWING OBJECTS TO BE QUERIED

SOPHIE CLUET[1]

INRIA, BP. 105, 78153 Le Chesnay Cedex, France

*April 8, 1998*

**Abstract** — This paper tells the story of OQL, the standard query language of the Object Database Management Group (ODMG) [30]. The story starts in 1988, at INRIA in the Altaïr Group.[†] The objective of that group was to develop an object-oriented database system [41]. This objective was reached: in September 1991 the $O_2$ database system started its commercial career as the main product of a company called $O_2$Technology [6]. As opposed to its competitors, $O_2$ featured a full-fledged query language named $O_2$SQL [22]. The story goes on with the creation of the ODMG in 1991 and the adoption of $O_2$SQL as the standard object query language under its new and final name: OQL. During the following years, OQL went through some modifications, the most important of which resulted in OQL 1.2 that offers some level of compliance with SQL92. On top of providing the expressive power of the SQL92 query language [54], OQL allows objects to be queried. This is a claim also supported by the upcoming SQL3. However, due to its adequacy to the object oriented type system and its functional nature, OQL is much simpler to learn, use and implement. A goal of this paper is to demonstrate this. This paper tells about the mistakes and pertinent choices we made while designing and implementing OQL. I hope it also conveys the great pleasure I had to be part of this adventure.

*Key words:* Object-oriented database, query language

## 1. INTRODUCTION

Among the half dozen commercial Object-Oriented Database Systems (OODBS) [4, 6, 7, 9, 11, 13] that appeared at the begining of the 1990's, only $O_2$ featured a full fledged query language. This may seem as a step backward in Database History. Had OODBS developers forgotten about the relational revolution, the famous physical/logical independance supported by query languages?

The answer is obviously no. One aim of an OODBS is to avoid the no less famous impedance mismatch encountered when programming a database application in, e.g., Cobol or C with embedded SQL. The Object-Oriented (OO) programming languages they support allow the writing of full database applications, involving data types richer than the flat relational tables, in a clean, modular and reusable way.

In that context, why did $O_2$ proposed both OO programming and query languages? The reason is simple: we believed (and were proven right) that we could have the best of both worlds. (1) The Turing complete OO programming language is the perfect tool to write complex applications. (2) The query language provides an easy and alternative way to write simple programs, or parts of programs, that can be optimized, and to query the database through interpreted *ad hoc* queries. The impedance mismatch is avoided by the fact that query and programming languages share a unique type system.

Although $O_2$SQL was the only commercialized query language, many others (e.g., [17, 18, 25, 29, 36, 46, 58, 61, 62]) had been proposed at that time in the literature, with or without prototype implementation. This lack of industrial competitors may be the reason why $O_2$SQL became the OQL standard. I prefer to believe that the main reason is its elegant design, the simplicity offered by its functional nature and its perfect compatibility with the OO type system. But then, I may be partial in that matter...

[†]Altaïr was a consortium funded by IN2 (the computer subsidiary of the Intertechnique group), INRIA (Institut National de Recherche en Informatique et Automatique) and LRI (the research lab of the University of Paris XI, Orsay). It was created for five years starting from September of 1986.

The story I am about to tell has two periods. The first period (1988-1991) witnesses the genesis of OQL, the second period (1991-1997) its evolution as an industrial product. But, before I get to the heart of the matter, let me set the stage.

## 2. CURTAIN-UP

### *2.1. The First Characters on Stage*

Two people are on stage when the curtain rises: François Bancilhon and Claude Delobel. They pull me along.

François is the charismatic founder and leader of Altaïr, a five year consortium whose goal is to develop an object-oriented database system called $O_2$. Later in the story, he becomes the no less charismatic president and manager of $O_2$Technology, the company which commercialized the $O_2$ system.

Claude is Professor in Computer Science at the university of Paris XI. He gives erudite advice to the Altaïr group. Later on, he participates in the creation of $O_2$Technology and becomes the company scientific adviser.

François and Claude are looking for a PhD student to work on a query language for $O_2$. I am a late student who happens to be here by chance, having followed my university team mate Jean-Claude Mamou. They offer me the job. I choose to forget that Claude is responsible for my worst grade in the Master program (Advanded Database Systems!) and I accept.

### *2.2. The Scenery*

When the story starts, the Altaïr group is one year old. The first $O_2$ data model has been designed [51] and its second and final version is well under way [49, 50]. Of course, Altaïr is not the only group proposing an object model: each OODB project has its own (e.g., [24, 29, 43, 52]).

However, there is more or less a general consensus on what should be an OODBS [19]. The various data models are similar, their differences comparable to those existing between two native versions of SQL. As a consequence, the Object Database Management Group (ODMG), created later on, will not have too much of a problem coming up with a standard data model acceptable by all.

In order to avoid a lengthy change of scenery between the first and second acts, I use the ODMG data model as unique background and introduce some $O_2$ specific features when needed. This does not alter the story and simplifies understanding.

#### *The ODMG Data Model*

**Note:** Since I am limited by space, my presentation of the ODMG model is simplified. First, I use the 1.1. release [30]. Other features can be found in [32] but they do not really influence the query language. Also, I use a rather abstract syntax that is neither ODL (the ODMG data description language) nor C++, Smalltalk nor Java (the three programming languages for which ODMG has defined bindings). Still, it borrows a lot from ODL and uses keywords found in the ODMG standard book [32].

Whereas a relational database consists of relations, an ODMG database is composed of objects and literals (also called immutable objects) which are categorized respectively into object types (also called classes in, e.g., C++ and $O_2$) and literal types. I define objects/literals and their types below along with their similarities and differences. Next, I define schemas. Finally, I present an example schema on which I will rely to present OQL.

1. Both object and literal types define the range of states of their instances.

   The state of an object or of a literal may be arbitrary complex, built in a recursive way from atomic (Integer, Float, Boolean, Character), pre-built (e.g., String, Bitmap, Date) or named types using two kinds of constructors: structure (a fixed number of named slots, also called attributes) and collection (Set[†], List[‡], Bag [§] or Array[¶]). In the $O_2$ data model [49, 50, 51], structures are called **tuples**.

Assume that *City* is the name of an object type already defined, then:

```
Structure(name: String,
          address: Structure(number: Integer, street: String, city: City),
          facilities: List(String))
```

defines a range of states among which we may find:

```
Structure(name: ``Hotel Louvre'',
          address: Structure(number: 14, street: ``Louvre'', city: ci1),
          facilities: List(``Bath'', ``TV''))
```

I will explain later the meaning of the `ci1` symbol that instanciates the *city* attribute.

In an object type definition, as opposed to literal, one usually omits the **Structure** keyword. The fact that the type is a structure can be infered easily: it is defined by a list of attributes and their types.

Also, in the representation of objects, I use from now on the following shortcuts: [] for Structure, {} for Set, ⟨⟩ for List or Array, and {{ }} for Bag.

2. Objects have an identity, literals do not.

   The identity of an object distinguishes it from all others. It is independent of its state: an object keeps its identity no matter what changes one performs on its state. Object identity also provides the means to reference objects and thus allows one object to be shared by others. To illustrate this, consider the previous example: `ci1` represents an object of type *City*. This object can be referenced by other objects or literals. If its state changes, the above structure will still reference the same object, but with a modified state. So will all other objects referencing it.

   Literals do not have an identity: they are characterized by their state. Accordingly, changing the state of a literal creates a new one. A given literal cannot have its state change (that is why literals are "immutable" objects). Further, literals cannot be shared (no referencing mechanism). For instance, suppose the facilities of a hotel are represented by the literal `List(``Bath'', ``TV'')`. One may assign a new value to the *facilities* attribute by, e.g., adding another string to the list. This does not, however, modify hotels that had similar facilities because there is no sharing.

3. Structured Objects support the notion of relationship. Literals do not.

   For example, in the two following object type definitions:

```
class Country name: String,
              cities: Set(City) is inverse of City::country

class City name: String,
           country: Country is inverse of Country::cities
```

   a relationship is defined between *Country* and *City* objects through their *cities* and *country* attributes. The consistence of this relationship is maintained by the database system.

   Note that I use the keyword **Class** instead of **Object Type**. In the sequel, I use one or the other interchangeably.

[†] A set is an unordered, duplicate free, collection of arbitrary cardinality.
[‡] A list is an ordered collection of arbitrary cardinality.
[§] A bag is a unordered collection of arbitrary cardinality.
[¶] An array is an ordered collection of fixed cardinality.

4. An object type also defines the behaviour of its instances. Literal types do not.

   The behaviour is defined by a set of operations.

   For example, the following definition associates the operations *add_city* and *rm_city* to the objects of Class *Country*.

```
class Country name: String,
              cities: Set(City) is inverse of City::country

          operations:
              add_city(City): Country
              rm_city(City): Country
```

   The implementation of these two functions may be stored in the database or not. Operations are sometimes called **methods**. In the sequel, I use one word or the other interchangeably.

   Although this is not part of the ODMG 1.1 version, let me now introduce the notion of encapsulation since, as we will see in the next section, it has caused me some embarrassment.

   The OO paradigm relies on classes having public and private interfaces. In Smalltalk, the state of an object is private: only the operations defined on a class can access the object's state. In other words, the state of an object cannot be viewed, queried or modified by anything but the operations that have been defined on its class. In other languages (e.g. $O_2$C, C++), the user may define its own notion of encapsulation and, for instance, define the whole or part of the object state as beeing public (i.e., readable, writable by all).

   Since, there was no common agreement on the subject, it has been left out of the first ODMG standard. We will see later on how OQL copes with encapsulation.

5. Object types may inherit from other object types, literal types cannot.

   The ODMG supports multiple inheritance: a user may define a class as being a subclass of one or more other object classes. A subclass inherits range of states (including relationships) and behaviour from its superclass(es). Moreover, an object of a class can be considered as an instance of any of its superclasses.

   Consider, for instance, the following definitions:

```
class Hotel name: String,
            address: Structure(street: String, city: City),
            facilities: List(String))
            category: Integer,
            price: Float
      operations:
            check_price(): Boolean

class Monument name: String,
               address: Structure(street: String, city: City),
               description: Text
      operations:
            change_description(Text): Monument

class Historical_Hotel: Hotel, Monument
      operations:
            check_price(): Boolean
```

   Class *Historical_Hotel* is defined as being a subclass of both classes *Hotel* and *Monument*. It inherits all attributes and operations from them. Thus, an object of class *Historical_hotel* has the following attributes: *name*, *address*, *facilities*, *category*, *price* and *description*. It also supports two operations: *check_price* and *change_description*.

Note that some attributes are common to both classes (e.g., *name*). Since their type is identical, there is no conflict and the definition is valid. If Class *Monument* had defined the *address* attribute as being of type String, a conflict would have been detected†. On the other hand, if *address* had been defined as:

```
address: Structure(street: String, city: Capital)
```

with *Capital* a subtype of *City*, it would have been correct, since an object of the former can be considered as an object of the latter. The infered type of the *address* attribute would then remain:

```
address: Structure(street: String, city: City)
```

since it fits both definitions.

Note also that the *check_price* operation appears in the definition of class *Historical_Monument*. This corresponds to a redefinition of the operation (i.e., a change of its implementation). When redefining an operation, the user may add parameters and/or refine existing ones. For instance, we could redefine the method *change_description* in class *Historical_Hotel* and have it return an object of class *Historical_Hotel* (instead of *Hotel*).

ODMG 2.0 supports a predefined class named **Object** which is the root of the class hierarchy† defined by inheritance (i.e., **Object** is a superclass of all other classes).

Finally, let me stress the fact that subtyping as defined by inheritance (i) concerns only object types and (ii) is explicit. The user has to define it, the system just checks the consistency of the specification. The $O_2$ data model, as well as other object data models (e.g., [29]), also supports an implicit partial ordering of literal types that relies on inheritance and a standard notion of subtyping. Roughly, a literal type is a subtype of another if it is more specific (e.g., one more attribute in a structure, a set defined on a subclass). In the ODMG, this weaker notion of subtyping is called type compatibility.

6. Objects types may have extents, literal types cannot.

   The extent of a class is the set of persistent objects belonging to that class. Once a class has been defined with extent, the system is in charge of managing the set of all its persistent instances. There are two disjoint ways to implement persistence in the ODMG: explicitly or by reachability. With the former, the programmer creates (resp. deletes) a persistent object by explicitely attaching (resp. removing) it to (resp. from) a database. It can be implemented efficiently from a system point of view but puts an extra load on the programmer. Specifically, the user is in charge of avoiding dangling references, i.e., references to deleted objects. Explicit persistence is well adapted to the maintainance of class extents. Persistence by reachability is not. This is explained now.

7. Both objects and literals can be named.

   By giving a name to an object or a literal, one makes it persistent as well as all objects and literals that it references directly or indirectly. This is called persistence by reachability. It requires the implementation of a garbage collector (GC) that deletes objects not referenced anymore.

   For instance, the following definition creates the names *Cities* and *Hotels*.

   ```
   name Cities: Set(City)
   name Hotels: Set(Hotel)
   ```

†These conflicts are usually solved through a renaming of the attributes, either implicitely or explicitely.
†Due to multiple inheritance, the class "hierarchy" is in fact a semi-lattice

The application is in charge of updating the names. By putting all created cities (resp. hotels) in the set *Cities* (resp. *Hotels*), the program will maintain extents. However, note that removing one city from the set *Cities* will not necessarily remove it from the database. Indeed, if e.g., an hotel references that city, it will persist. Otherwise, it will be garbaged (i.e., deleted).

Persistence by reachability avoids having to deal with dangling references, i.e., references to objects that are not part of the database. However, since it relies on a GC, it makes it difficult to support automatic extents. Indeed, a transaction may commit before all objects have been garbaged in which case the extent may contain non-persistent objects. We will see that this is an important problem since extents are needed by the optimization process.

In ODMG 2.0, both Java and Smalltalk bindings support persistence by reachability but no extents. On the contrary, the C++ binding supports explicit persistence and extents.

An ODMG schema is composed of (i) a set of object and literal type definitions, (ii) a class hierarchy and (iii) a set of names. Access to the database is gained either by names or by asking for the extent of a class. Both methods are similar. In the sequel, I will forget about class extents since the original $O_2$ system relied on names.

Figure 1 summarizes the main differences between relational and ODMG models.

| | **Relational Model** | **ODMG Model** |
|---|---|---|
| **Entry Points** | Relations | Names<br>or Extents |
| **Persistence** | Membership in a relation | Attachment to a name + GC<br>or Explicit persistence |
| **Types** | Flat relations | Complex types |
| **Operations** | No | Part of the schema, maybe of the DB |
| **Keys** | Yes | Yes in Version 2.0 |
| **Integrity Constraints** | Referential integrity | Relationships and keys |
| **Applications** | SQL + a programming language | C++ and optionally OQL<br>or Smalltalk and optionally OQL<br>or Java and optionally OQL |

Fig. 1: Differences Between Relational and Object Models

*An Example Schema and Database:*

Figure 2 presents a simple object schema with very few methods. It is sufficient background for the story. It features five classes, three of which are related by inheritance, and six names. Figure 3 shows a partial instance of this schema.

This example was our favorite in the Altaïr days. I must say I am rather happy to travel on these shores again.

## 3. AND THE STORY STARTS: THE GENESIS

This section covers the first period of the story of OQL, from June 1988 to September 1991. It describes the design and implementation of the $O_2$SQL language in the Altaïr group.

### *3.1. The Language Forebears*

The concept of relational database was introduced in 1970 by Codd [38], that of object-oriented database by Copeland and Maier in 1984 [40]. Between these two dates, a lot of work was done

| **Classes** | |
|---|---|
| Country | name: String<br>cities: Set(City) **is inverse of** City::Country<br>currency: struct(name: String, rate: Float)<br>**Method** Add_city(City): Country |
| City | name: String<br>country: Country **is inverse of** Country::cities<br>map: Bitmap<br>hotels: List(Hotel)<br>monuments: Set(Monument) **is inverse of** Hotel::city |
| Hotel | name: String<br>address: struct(street: String, city: City)<br>facilities: List(String)<br>category: Integer<br>closing_days: Set(String)<br>price: Float<br>**Method** check_price(): Boolean |
| Monument | name: String<br>address: struct(street: String, city: City)<br>description: Text<br>closing_days: Set(String) |
| Historical_Hotel | **Inherits** Hotel, Monument<br>**Method** check_price(): Boolean |

| **Names** | **Types** |
|---|---|
| Countries | List(Country) |
| Cities | Set(City) |
| Paris | City |
| Hotel_of_the_Month | Hotel |
| Hotels | Set(Hotel) |
| Monuments | Set(Monument) |

Fig. 2: An Object-Oriented Schema

| | | |
|---|---|---|
| **Countries** | = | ⟨ c1, c2, ...⟩ |
| **Cities** | = | {ci1, ci2, ... } |
| **Hotels** | = | { h1, h2, ... } |
| **Paris** | = | ci1 |
| **Hotel_of_the_Month** | = | h1 |

....

**c1** : [name: "France", cities: {ci1, ci2, ...}, currency: [name: "FF", rate: 1] ]

...

**ci1** : [name: "Paris", country: c1, Map: m1, hotels: <h1, h2, ...> ]

...

**h1** : [name: "Hotel du Pont Neuf", address: [street: "Malaquais", city: ci1], ...]

Fig. 3: A Partial Instance

to extend the relational model, notably by removing its first normal form[†]. These efforts resulted for the most part in two main families of models, nested-relation (e.g., [53, 59, 63]) and complex-object (e.g., [20, 48]), both of which featured algebra-based (e.g., [15, 28, 39, 47, 56]), calculus-based (e.g., [16, 23, 47]) and SQL-like (e.g., [57, 59]) query languages.

Thus, when we began to work on the $O_2$ query language in 1988, we had a lot of material to start from. I do not wish to compare $O_2$SQL to all existing query languages, I simply cite here the languages I remember as being our main sources of inspiration. There are mainly four:

- SQL [54], from which we borrow some syntax,
- the algebra/calculus for complex objects of Serge Abiteboul and Catriel Beeri [14], the Col Datalog-like language for complex objects of Serge Abiteboul and Stéphane Grumbach [16], and the OSQL object query language of David Beech [25] from which we borrow the functional way to construct nested structures.

## *3.2. A first Experience*

After having read a lot of papers related to the topic, I spent six months with Claude Delobel and two other Altaïr fellows, Christophe Lécluse and Philippe Richard, designing and prototyping an SQL-like query language called Reloop [36] whose semantics relied on an algebra. Reloop borrowed much from [25].

The language was rather nice, but had two major weaknesses, common to all its rivals: only collections could be (i) queried and (ii) returned as an answer. Obviously, the language inherited some SQL specificities that were not appropriate in the object context. Indeed, the object model supports any kinds of objects. One would like to be able to query them (e.g., to know the name of the *Hotel_of_the_Month*), to construct and return them (e.g., assign a new hotel to the name *Hotel_of_the_Month* using a query).

## *3.3. The Good Idea*

Then, François Bancilhon came up with a good idea: "the language should be functional". Types are constructed in a functional way by combining atomic and named types using constructors (e.g., tuple, set). We should build queries the same way.

So, we threw everything away and started anew, all convinced that this was the right way to go. On top of being easy to learn and implement (although implementing it efficiently is another story), a functional query language is easy to design. One just has to associate to each atomic type and constructor a set of basic queries. Object types add their own (their set of defined operations). Then, queries are built by combining basic queries in the same way types are combined. There is just one restriction: the query should be well typed, e.g., you cannot combine a set input query with one outputing a tuple.

Let me illustrate this. One starts building queries with base queries of arity 0.

**Example 1** *Constant Queries*
2
"Paris"
Paris

are examples of these. Their result should be rather obvious. They return, respectively, 2, "Paris" and **ci1** which is the object associated to the name *Paris*.

Then, one combines queries according to types.

**Example 2** *Simple Combinations*
2 + 2
"Paris" + "sur Seine"
Paris.name

[†]The first normal form requires atomicity of the attributes of a relation

**count**(Cities)
are examples of queries obtained by simple combination. Note that the '+' operator (basic query) is overloaded: it denotes integer addition in the first query and string concatenation in the second. It can also be used to denote union of sets/bags or list concatenation. The '.' operator is used to extract one attribute from a structured object[†] or literal, *count* can be used to count the elements of any collection: in that case a set of cities.

I guess the basic principle is understood at that stage, let me just go one step further.

**Example 3** *Going Further*
Paris.name = "Paris"
Paris.hotels[0]
**tuple**(cityname: Paris.name, countryname: Paris.country.name)
Here, the first query returns the boolean **true**, the second extracts the first element in the list of Parisian hotels, the last constructs a structured literal (remember that $O_2$ used the keyword **tuple** to denote structure), the second attribute of which is specified through a path expression that first extracts the country associated to Paris and then the name of that country.

Most database programmers would fail to see a query language in these examples. Indeed, an essential feature is lacking: associative access to the database. This leads us to the second choice we made.

## *3.4. Our First Disagreement: a Syntactical Choice*

As far as operators for atomic and tuple types were concerned, we did not have any problem in coming to an agreement. This harmonious climate came to an end when we started thinking about the way to define associative access: i.e., iterators.

François was very much in favour of a functional syntax. Claude and I were convinced that the SQL-like syntax was more appropriate. It is interesting to note that our motivations were purely aesthetic. François thought that adopting the full functional contraption worked for a clearer language, Claude and I believed that a *select-from-where* statement was easier to write and read and could be learnt faster. We did not think at all about compatibility with SQL. I guess the reason was that, at that time, no one was bringing the two technologies together. Relational systems were yet purely relational and object systems were targeting new applications (such as CAD/CAM, GIS, etc.). Thus, we did not see any reason to consider communication between the object and relational worlds.

So, eventually, after much fighting, we chose the SQL syntax. *A posteriori*, this was a smart choice. I did not keep track of the functional syntax on which we debated. Still, I think the following is a good approximation.

**Example 4** *What You Missed and What You Got*
The query returns the names of the countries having a currency whose rate is bigger than 2 (as compared to French Francs).

| | | |
|---|---|---|
| **list** ( c.name; c **in** Countries; c.currency.rate > 2 ) | **select** | c.name |
| | **from** | c **in** Countries |
| | **where** | c.currency.rate > 2 |

This illustrates the fact that the choice was indeed purely syntactic. Both syntaxes contain the same information about the expected result. The only difference is that the functional syntax states that we are iterating over a list: the list of countries. The SQL-like syntax relies on type inference to obtain this information. Both queries return a list of strings whose order reflects that of the input list

I will come back to typing issues. For now, just note the syntactic difference between SQL and $O_2$SQL **from** clauses. $O_2$SQL uses the **in** keyword to define the range of a variable. As we will

[†]Two syntaxes are available to extract attributes from structured objects: '.' or '->'.

see in the second part of this paper, this has changed. OQL 1.2 accepts both SQL and $O_2$SQL syntaxes. I must say I rather prefer $O_2$SQL's which, I believe, is more natural (*c* is in *Countries*) and works for a clear separation of the various parameters when complex queries are involved. We will see that later on.

**Example 5** *More Queries*

The first query selects the French hotels that are overly expensive in comparison to what they offer. For each such hotel (the chances are that there will be some), it constructs a tuple with two attributes: the name of the hotel and a boolean stating whether the hotel provides individual bathrooms.

```
select   tuple( name: h.name, bathroom: "bath" in h.facilities )
from     h in Hotels
where    h.address.city.country.name = "France" and not(h->check_price())
```

Consider the **select** clause. It features a query that constructs a tuple which itself combines two queries: a simple path expression and a boolean query (**in**) applied on another path expression. Now, the **where** clause consists of an **and** query, which combines two boolean subqueries. The first is an equality test between a long path expression and the constant "France". The second query combines a **not** with a boolean method call.

Remember the query of Example 4: it iterated over a list and featured a string query (*c.name*) in its **select** clause. Thus, it returned a list of strings. This one iterates over a set and features a tuple query in its **select** clause. Yet, it does not return a set of tuples, but a bag. The reason is rather obvious. The query may generate duplicate values (if two hotels have the same name and bathroom facilities) and duplicates represent meaningful information. If one wants to avoid duplicates, the **distinct** function transforms a bag into a set.

The distinct function was used like any other function when we first designed the language. I.e., it was applied on the result of the iteration in the following way:

**distinct (select** ... **from** ... **where** ...**)**

Before $O_2$ became a commercial product, this was changed to:

**select distinct** ... **from** ... **where** ...

as in SQL. I think we owe this first SQL-oriented change of syntax to Guy Ferran who we will meet soon.

But let me come back to combining queries. The next query compares the hotel that has been selected as *Hotel_of_the_Month* (see the schema) with the other hotels. It returns a unique tuple with two attributes: the hotel of the month and a set of hotels having at least the same number of facilities and a cheaper price.

```
tuple(   hotel:            Hotel_of_the_Month,
         cheaper_hotels:   select   distinct h
                           from     h in Hotels
                           where    count(h.facilities) >= count(Hotel_of_the_Month).facilities
                                    and h.price < Hotel_of_the_Month.price )
```

This query illustrates the fact that an iterator can be combined as any other query. Let me illustrate that point further by combining several iterators, one of which is a newcomer. The following query returns, for each city with at least one monument open on Sunday, their name and 3 stars hotels.

```
select   tuple ( city: c.name, 3hotels:   select   h
                                          from     h in c.hotels
                                          where    h.category=3 )
from     c in Cities
where    exists m in c.monuments: not ("sunday" in m.closing_days)
```

First note the nested iterator in the **select** clause that associates to each city its list of 3 stars hotels (*c.hotels* is a list query, see the schema). A similar iterator could occur in any clause. Also, note the **exists** boolean iterator in the **where** clause. It has a universal counterpart: **forall**. These iterators have two clauses: one stating the quantified collection, the other the associated boolean. Again, both clauses accept any subquery of the correct type. Here, the queries are simple. They can be arbitrarily complex.

Finally, let me combine queries in the **from** clause. As in SQL, we may define queries over several collections. The following one returns the names of the hotels of Nice.

```
select   h.name
from     c in Cities,
         h in c.hotels
where    c.name = "Nice"
```

Note the relationship between the two variables of the **from** clause. The query defining **c** is constant, that defining **h** depends on **c**. This allows navigation inside nested collections: here, a set of cities, each of which has a list of hotels. So, what does this query that iterates over both un-ordered and ordered collections return? It returns a bag of strings. The reasons for this is that it is easy to convert a list into a bag and impossible to convert a bag into a list in a deterministic way. Actually, we will see that OQL implicitely converts all input collections into bags. E.g., as opposed to what I presented here, iterating over a single list breaks its order. I much prefer this new way of typing iterators: it is simpler and lends itself to more efficient implementations.

The last query of this presentation, is equivalent to the above one.

```
select   h.name
from     ci in   (select   c
                  from     c in Cities
                  where    c.name = "Nice")
         h in    ci.hotels
```

Note the nested iterator in the **from** clause. It is constant, but we could have one depending on some previous variables. Now, why should anyone write a query like this when other simpler ways are available? My experience with $O_2$ is that such queries are rarely written directly, but are frequently generated by programs.

At this stage, the SQL connoisseur must have noticed that two clauses are missing: **group by** and **Order by**. This brings us to what I believe was our first mistake.

## *3.5. Our First Mistake*

Grouping and sorting have their own iterator in $O_2$SQL. A **select-from-where** iterator supports only three clauses. I cannot remember why we were so keen on having separate operators for grouping and sorting. I think the idea was that we wanted to be able to perform these operations directly on any collection. Retrospectively, I believe this was a mistake. It has been corrected in OQL 1.2 and I must say that I much prefer the SQL-like syntax after having voted against it.

Anyway, the two following examples show how these operations were performed in $O_2$SQL and OQL 1.1.

**Example 6** *Grouping and Sorting*

This query groups the hotels according to their price. It then returns for each price, the set of hotels with less than 3 stars and whose price is correct.

```
group all   h in Hotels
by          price: h.price
with        hotels:        select   h
                           from     h in partition
                           where    h.category < 3 and h->chech_price()
```

Note the use of a variable called *partition* in the nested **from** clause. This variable is system-generated and works on the partitions created by the grouping operation. If we remove the optional **with** clause in the above query, it returns a set that could look like:

```
{ [price: 500, partition: {{ h2, h3, h5 }}],
  [price: 550, partition: {{ h4, h7, h8 }}],
  ....
}
```

Using the **with** clause, the user may then work on each separate partition and, for example, apply methods on its elements (as is done above). This results in the removal of the *partition* attribute which is replaced by those defined by the user. In that case, the result could be:

```
{ [price: 500, hotels: {{h2, h5}}],
  [price: 550, hotels: {{}}],
  ....
}
```

Note that this is different from the SQL semantics of grouping. We will discuss this issue further in the second part of the story.

The **group** operator is in fact a goodie. For example, using nested queries, one can formulate the above query in the following way:

```
select   distinct tuple ( price: h1.price, hotels:  select  h
                                                   from    h2 in Hotels
                                                   where   h2.category< 3 and h2->check_price()
                                                           and h1.price = h2.price )
from     h1 in Hotels
```

The next query sorts the set of cities on their country and number of monuments. It returns a list of cities.

```
sort all   c in Cities
on         c.country, count(c.monuments)
```

Note that the second sorting criterion is a bit unusual. Again, as is the case for all iterators, one may use any query in any clause as long as typing is correct. The grouping/sorting iterators only require a collection subquery in the **group/sort all** clause and atomic or object subqueries in the **by/on** clause. There is no type restriction on the **with** clause of a grouping operation.

Let me go on now with a funny anedocte concerning sorting.

## *3.6. Trying to Avoid Non Determinism*

Does the above query strikes you as being non-deterministic? Well, it is. One can imagine two cities of the same country with the same number of monuments. The query does not say which one should come first in the resulting list. This does not trouble you? Actually, I do not remember why it bothered me so much, but it did.

To solve this capital problem, we came up with a brilliant idea: the **magic** function that associates a distinct number to all the elements of a collection. By adding implicitely a call to that function in all sorting operations, we removed non-determinism.

Or so we thought at some point of our designing phase. We just forgot about the non-determinism inherent to run-time errors in a declarative language (to be discussed soon). I guess the reason of this lapse was that the prototype I was implementing at that time was intended as a means to validate the design and did not support any optimization: thus, until the end of 1989 the language was declarative only in name.

## *3.7. But Forgetting Run-Time Errors...*

$O_2$SQL can be typed statically thus removing many potential errors. However, as in most languages, run-time errors are possible. They are the following:

- Division by zero.

- **min/max/avg** operations on an empty collection. Note that this is different from SQL that relies on Null values to avoid run-time errors. Although the main reason for this difference is the absence of Nulls in the $O_2$ model (as well as in the ODMG model first releases), I must say that some odd behaviour generated by Nulls makes me wonder about the best solution (see Section 4.4).

- **element** operation on a collection with more or less than one element.

  **element** is an operation for extracting the element, that we know to be unique, from a collection. For instance, it may be used to extract a specific hotel from the set of hotels in order to, e.g., assign a new object to the *hotel_of_the_month*.

  ```
  element   ( select   h
              from     h in Hotels
              where    h.name = "Crillon" and h.address.city.name = "Paris" )
  ```

  If there are several "Crillon" hotels in Paris, the query will generate a run-time error.

- Positional operations on lists and strings.

  $O_2$SQL supports a number of operations allowing to extract a sublist or an element from a list. These operations are also available on strings which are considered as lists of characters. I presented one such operation in Example 3. Another example is:

  Paris.hotels[0:2]

  which returns the sublist of Parisian hotels starting from the first and ending at the third (we use the C language way of interpreting positions in an array).

- All operations but equality test on a **Nil** object.

  The ODMG model of [31] does not support the **Null** value as found in SQL. Objects alone can be undefined. This is represented by **Nil**. As to literals, they support default values (e.g., the empty list, the empty string, the 0 integer, etc.). Trying to get some information out of the **Nil** object results in a run-time error.

So, why do run-time errors add non-determinism to declarative languages? Consider the following query:

```
select   h
from     h in Hotels
where    h->check_price() and h.address.city.name = "Paris"
```

Assume there exists one hotel whose city is undefined, i.e., *h.address.city = Nil.* Then, evaluating *h.address.city.address* generates a run-time error. However, note that if we evaluate the boolean operations in the order given by the user and if the concerned hotel does not have a correct price, the query can be evaluated without generating any error. The problem arises only if we consider re-ordering the predicates which should definitely be possible since the language is declarative: i.e., no algorithm is specified by a query, only a result; the system is in charge of finding the best algorithm.

Run time errors are actually quite bothersome in a declarative query language since, as opposed to imperative languages, one cannot remove them entirely. We will see how we improved things later on in the story.

### *3.8. And Method Calls!*

Another cause of non-determinism is related to method calls and is much more troublesome since methods can actually change the state of the database. Again, consider the above example and suppose that the *check_price* method updates a persistent object which keeps tracks of the badly rated hotels. The state of this object obviously depends on the algorithm that will be chosen to evaluate the query.

The only solution to that problem relies on forbidding the use of methods having side-effects in the query language. However, note that implementing this solution is not trivial. Indeed, one can come up with some sufficient conditions to eliminate methods with potential side-effects but (i) we may wrongly classify a method as dangerous and (ii) maintaining this information is very costly. An alternative and reasonnable solution consists in asking the user for that information as in [60]. I am sorry to say that this problem is not resolved in the ODMG standard. In $O_2$, we relied on a warning to users.

## *3.9. Brief Overview of the Language Semantics*

The informal presentation of $O_2$SQL is over. Many basic queries are missing (see [21]) but the general idea should be understood. It is not my intention to formally define here the semantics of the language, this can be found in [21]. Still, I want to insist on the fact that $O_2$SQL, as opposed to e.g., SQL, has a full formal semantics. Moreover, due to its functional nature, the language can be easily and formally extended.

Briefly, $O_2$SQL is a first order functional language. It is defined by a set of basic functions/queries and a way to build new ones through composition and iterators. Defining the set of basic queries is rather trivial. Composition comes for free. The only difficulty has to do with defining formally the iterators. This was done in a manner resembling list/set comprehensions (e.g., [42]).

Now, having functional semantics does not mean that only functional optimization can be considered. Indeed, as we will see in the sequel, one can use an algebra to implement $O_2$SQL iterators. The only requirement is that this algebra features an operation to apply a function on each element of a collection. This operator exists under various names in [14, 26, 35, 45, 61]. Also, the literature now offers optimization formalisms relying on a functional approach [42, 34].

## *3.10. First Public Presentation*

Here are my memories of the reactions generated by our first public presentation of the language.

The date was June 1989, the place is Salishan Lodge, Oregon, where the second International Workshop on Database Programming Languages (DBPL) was held. I was to give my (first) talk on the second day of the workshop and was very worried: on the first day, one talk had generated such controversy that the speaker did not have the opportunity to finish. And I was right to worry. The uproar started as soon as I said that $O_2$SQL violated encapsulation.

Remember what I said in the presentation of the ODMG model about public and private properties. In a "pure" object-oriented environment, the state of an object is private. The idea of encapsulation is that the methods/operations defined on a class are the *only* way to operate on the objects of this class. This enforces good modularity of the code and good programming discipline. Furthermore, it allows the user to trace bugs in a simple way since a "bad" object state can only be due to a very limited set of operations.

If the query language cannot violate encapsulation, then basically the only access to objects is through methods and data is invisible. Conversely, if we want to see data, we must be able to violate encapsulation. Let me give you my current understanding of the subject.

**Why should objects be visible?** A first reason is connected to testing and debugging: to check that a program does what it is supposed to do, one needs to view the objects' state. Another reason concerns performance. Invoking a method can be expensive, especially if it has been redefined. It implies finding the appropriate method implementation at run time (late binding) and entails the preparation inherents to function calls. Furthermore, it is difficult to optimize a query featuring method calls. Thus, one would like to avoid having to call a method for every read access, as required if we strictly respect encapsulation.

**Why should objects be hidden?** I can mainly see two reasons: (i) to hide information from some people and (ii) to maintain a good programming discipline. However, (i) is a matter of access rights, not of encapsulation and (ii) should not influence the query language. A query language is not a programming language, it does not perform updates on the database†.

This issue looks a bit dated nowadays. There is not one line on the subject in the ODMG book and $O_2$ programmers have been violating encapsulation for years without complaining. Yet, at that time, it was very much debated and most query languages implemented encapsulation (e.g., [36, 52, 46]). As a matter of fact, we did think that encapsulation should not be violated when the query language was used as a programming tool and proposed two modes: (i) an interpreted *ad hoc* mode of direct interrogation where every object could be read and (ii) a programming mode that hid the objects states. We never implemented the second one but provided a function to allow the execution of an *ad hoc* query from within a program. The next release of $O_2$ will provide a query compiler that will violate encapsulation as much as its current interpreter.

Anyway, I eventually managed to end the hair splitting discussion on encapsulation by giving a rather rude French translation of this activity, and finished the talk on $O_2$SQL. I think the DBPL audience liked the language very much and that the functional approach was indeed viewed as a great idea. Peter Buneman added the icing on the cake: he invited me to join his famous "2+2 Query Fan Club".

## *3.11. Implementation*

Once the language was designed, implementation became the key issue. The first prototype I implemented relied heavily on the language functional semantics. Each basic query and iterator was implemented by a function, these functions were combined as queries are. No optimization was performed. Implementing the full language in this way required a little more than one month. Although the resulting software was obviously not intended to become an industrial product, it was an interesting experience since it highlighed two troublesome points in the implementation:

- The first one is related to typing. The language allows the construction of potentially empty collections. For instance:

  **Set**("France", "Italia", "Espagna")
  **Set**()

  are two queries returning sets. They can be useful to, e.g., test the set component of an object.

  Typing the first query is easy (Set(String)), typing the second is impossible if one does not know the context in which it is called. In order to type it free of context, we need a supertype of all object and literal types. This supertype exists in some theoretical data models (e.g., [17]), however, neither $O_2$ nor the ODMG supports it. Thus we have to rely on context and, from experience, it is rather easy but requires a lot of tedious code. I would say that one fifth of the type inference module is devoted to this specific problem.

- The second difficulty is related to encapsulation. As I explained before, $O_2$SQL violates this principle. This means that an object can be interpreted in two different ways. Let me illustrate that with the object **ci1** corresponding to Paris (see Figure 3). We can interprete **ci1** as an abstract entity to which we may apply methods. Alternatively, we can interprete it as the literal it encapsulates, i.e., as:

  [name: "Paris", country: c1, Map: m1, hotels: <h1, h2, ...> ]

  Again, there is nothing really complicated here but this duality of object interpretation requires significant coding.

  Actually, most of it was due to my beginner's zeal. Consider a membership test between the result of two queries. Suppose the first query is of type $t1$ and the second of type $set(t2)$. Now, if $t1$ is a non-strict subtype of $t2$, everything is fine, we can rely on some system primitive to evaluate the membership test. But, what happens if $t1$ is a literal type and $t2$ an object type. One could think that there is a type incompatibility. However, due to the dual

†Actually, updates can be performed through methods but we have seen that this was indeed a bad idea and that it should be forbidden.

interpretation of objects, one may also consider the literal type encapsulated in the object type *t*2. If it is compatible with *t*1, then the query can be evaluated even if the required conversion from objects to literals is rather costly.

Fortunately, Guy Ferran joined Altaïr in 1990, at the right time to put an end to my zeoletry. We restricted desencapsulation to the most obvious cases (e.g., access to the properties of one object) and introduced an operator to explicitely access the state of an object (i.e., **Paris* returns the state of the *Paris* object).

Guy is one major character of this story. He plays two parts. He is the technical manager of $O_2$ and was until 1993 the representative of $O_2$Technology at the ODMG. He is responsible for the adoption of $O_2$SQL as the ODMG standard.

Guy and I worked together on the second interpreter of the language. Claudia Roncancio joined us. The idea was to keep the functional semantics for all basic queries and to give an alternative algebraic semantics to the iterators (using an operation, called *Map*, to apply functions to the elements of a collection). We improved the previous prototype on primarily three points: (i) constant and common sub-expressions factorization, (ii) use of indexes, (iii) simple algebraic rewritings.

**Common subexpression elimination** is very important in the object-oriented context: a query may contain several occurences of potentially expensive sub-expressions (e.g., method calls) especially when we iterate over sets of object identifiers. Remember from my presentation of the ODMG model that objects have identifiers. Among other things, they allow information sharing. For instance, the object *c*1 is referenced by the collection denoted by the name *Countries* and by several *City* objects. One reference to an object is implemented using the object identifier. Thus, when we iterate on the set *Countries*, we simply load a set of identifiers. Sometimes, the state of the object comes along (e.g., when it is stored on the same page), but not always. Thus, getting simple information, such as the name of a country, may cause a page fault and entail an input operation. In that context, factorizing even very simple expressions is important.

**Indexes** in the object-oriented context come in different flavours [27]. Their main difference with relational indexes is that keys are given by path expressions, not by a single attribute. This means that one can use an index to evaluate an expression of the form:

c.currency.rate > 2

Apart from that, they are used to implement selections or joins as in relational systems.

Let me come back to typing issues. Remember that iterating over a list returns a list. The order of a list is given by the user. There is no way to reconstruct it logically. As far as we know, the objects in the list *countries* can be ordered according to names, creation dates or the tastes of the programmer. Now, consider Example 4 and suppose we want to use an index on the above path to evaluate it. An index implicitly orders the elements of a collection according to the key. In that case, it means that it does not preserve the list's logical order. In that context, unless we additionaly store the positions of the elements in the list (which is costly to maintain), we cannot rely on a simple index scan to evaluate selections on lists. This is the reason why, in OQL, we changed the typing rules of the **select-from-where** iterator: it always returns a bag, unless **distinct** is used in which case it returns a set.

**Algebraic Optimization** is used in the object context in a manner similar to the relational context. I will come back on that issue later on. At that time, the optimization we performed was trivial: we mainly pushed selections and transformed some joins into semi-joins.

So, this brief summary of our implementation efforts ends the first period of the story. A few months later, the $O_2$Technology company was created. $O_2$SQL second interpreter was part of the $O_2$ product and I am very proud of that.

## 4. GOING PUBLIC

$O_2$Technology was founded in 1991. The same year, I got a position at INRIA and joined the VERSO database group [1]. François proposed to take me as a technical adviser in the new company and I was happy to accept. I have been keeping an eye on the language since then. Soon afterward, the Object Database Management Group was created and Guy Ferran became the $O_2$Technology representative there. He is responsible for $O_2$SQL's proud new identity: OQL, the standard query language of the ODMG.

This section tells about the evolution of $O_2$SQL to OQL 1.1, and from there to OQL 1.2 and 2.0. I play an underground part in this story: I never went to any ODMG meeting nor met its foreign (understand non-French) gurus. My underground activity was twofold: (i) I was, until recently, responsible for the $O_2$OQL optimizer/interpreter, and (ii) I worked on the various OQL releases with Guy, François and a nice bunch of people. Let me introduce them by order of appearance on stage: Jim Melton and Jeff Mischkinsky from Sybase (OQL 1.2), Cassio Souza dos Santos and Sophie Gamerman from $O_2$Technology (OQL 2.0), Fernando Velez from Unidata (the upcoming OQL of which I will not speak).

Before I start, I must warn you that I am not impartial. This presentation reflects my feelings, not that of the ODMG members.

### *4.1. $O_2$SQL Becomes OQL*

In order to become OQL, $O_2$SQL took the ODMG look. This required only minor syntactic changes. One just has to replace the word **tuple** by **struct** to make all queries of Section 3 ODMG compliant. Also, do not forget that typing has been slightly changed (see Section 3.11).

Furthermore, we added two features that were lacking in $O_2$SQL: object creation and casting.

**Object Creation** was nearly as much a subject of dissention in the object database research community of the eighties as encapsulation.

People in favour argued that it was mandatory in order to obtain completeness and closure property. I fully agree on the matter of completeness and I think this was why Guy extended the language in that direction. The second argument seems more dubious: even without creating objects, $O_2$SQL covers the full object model and input and output data are of the same nature. In any case, very few languages supported object creation (e.g., [17, 61]), most did not implement it (e.g., [62, 46, 36]). As far as I understand, the lack of dynamicity in type creation was the main reason.

Relations can be created dynamically since their domain can be inferred from a query. But, an object type captures more than a domain: it comes with a behaviour (set of operations) and has to be situated in a class hierarchy. This cannot be inferred[†] and has to be specified by the user. Forbidding queries requiring type creation comes as a natural consequence of this lack of dynamicity. However, there is no good reason to forbid the creation of objects on some existing type. When first designing the language, we focused too much on typing issues and missed that truth. OQL 1.1 corrects the situation: it provides the means to create objects of existing types but forbids type creation.

Another reason for forbidding object creation is related to the way persistence was defined in some models (e.g., [24]): if one attaches persistence to types, creating an object affects the state of the database and we have seen in Section 3.8 that side effects are not good in a query language. However, this is not a problem with the ODMG model: types and persistence are orthogonal. To become persistent, an object or a literal has to be attached to some root of persistence. Indeed, consider the example database. A newly created *City* object will not persist unless we make it accessible from some persistence root by, e.g., adding it to the set named *Cities*. Thus, creating an object in a query has no more effect on the database state

[†]Some research work on object views proposed an automatic classification of object types based on structure. Used without caution, this may lead to, e.g., considering a person as a special dog.

than creating a literal. Again, if one uses a method having side effects we are in trouble but this is true with or without object creation.

Let me now illustrate the new feature with some examples. The first query creates an *Hotel* object.

```
Hotel(  name: "Medicis",
        address: struct(street: "Raspail", city: Paris),
        category: 3,
        price: 579)
```

Note the use of the word **struct** instead of **tuple**. This is part of the ODMG look I mentioned above. Also note that the query assigns values only to some attributes of the structured object. The missing *facility* attribute will be given a default value (in this case, the empty set). All attributes could be missing in which case a similar treatment will be applied to all. Finally, note that the created object is transient. If the query is part of a program that, e.g., assigns its result to some root of persistence, then it will become persistent.

Another way to create an object, is to give its full value. This is useful for non structured objects (e.g., sets), but can be applied in all cases. For instance, consider a class *CheapHotels* that has been defined as a set of hotels. The following query constructs an object of that class.

```
CheapHotels(  select  distinct h
              from    h in Hotels
              where   h.price < 300)
```

The state associated with the created object is the result of the embedded query.

As in any OQL query, a query creating an object can be the parameter of another query. For instance, consider the following class definition:

```
Class CityStat name: String,
               nb_hotels: Integer,
               avg_hotel_price: float
```

Then, the query below returns a bag of newly created *CityStat* objects:

```
select  CityStat(  name: c.name,
                   nb_hotels: count(c.hotels),
                   avg_hotel_price: avg(select h.price from h in c.hotels))
from    c in Cities
```

Let me conclude by pointing out a missing feature. OQL does not allow the creation cyclic objects. This would require a way to name a newly created object so as to be able to reference it in the same query. For the moment, cyclicity in object creation is managed by the application program.

**Casting** adds flexibility to static type checking. Consider the following query:

```
Element  ( Select  h.description
           From    h in Hotels
           Where   h.name = "Crillon" and h.address.city.name = "Paris" )
```

A static analysis will detect a type error since *description* is not an attribute of Class *Hotel*. Yet, the user may have the knowledge that the "Crillon" hotel is in fact an object of Class *Historical_Hotel* with a *description* attribute and that, accordingly, no type error will occur at run-time. However, run-time type checking is not efficient since it entails an analysis of each element of the collection on which we iterate.

Casting operations combines dynamicity with the efficiency of a static analysis. Of course, this may lead to run-time errors. This may occur when one relies on wrong semantic information, but also may occur because the user cannot be sure of the way the query is going to be evaluated. For instance, consider adding a casting operation to the above query.

```
Element   ( Select   ((Historical_Hotel) h).description
            From     h in Hotels
            Where    h.name = "Crillon" and h.address.city.name = "Paris" )
```

In that case, the chances are that the casting operation will be evaluated after the selection on *name* and *address*. Thus, if the user assumption is correct, the query will pass evaluation. Now, if we use casting in the **where** clause or in more complex queries, we may run into problems. A good implementation can avoid most, but not all. I have to admit that I do not like casting. Yet, I believe it is an essential feature in the OO context that, as in OO programming languages, has to be used with great caution.

## *4.2. OQL Meets SQL*

In 1994, members of the ANSI X3H2 and X3H7 committees (in charge of defining SQL3) met members of the ODMG. I think the main idea was to see how both standards could fit together on the market place. Alas, I was not part of this meeting which I am sure was a lot of fun. But I attended another and more informal one on the same subject. Jim Melton (who is the SQL editor) and Jeff Mischkinsky (from Sybase) came to France in April of 1995. They had the good idea to take advantage of this visit to continue the ANSI/ODMG discussion with François and Guy. I was in the area and managed to get an invitation. I describe here how this meeting influenced OQL. For potential influences on SQL3, you may read Jim's account [55].

The first idea was to see if, by adding a type relation defined as a **bag of struct** in the ODMG model, one could merge the two languages. It is funny to see how some ideas that seem so bright at one moment look wholly fuzzy some years later. Among them, my favorite concerns the way we thought this merge could be done: either with one syntax and two languages or with two syntaxes and one language. Clear, isn't it? Anyway, as far as I am concerned, the main result of this discussion is not a merge of the two languages but some modifications to OQL that, for the most part, improve the language and give it some level of compliance with SQL-92: SQL queries can be translated very easily in the new OQL, some queries do not even require a transformation. This is indeed useful to implement an ODBC interface on top of an object database system but, also, to fool naive users in making them believe they are programming in SQL (just kidding, I think this is a terrible idea). Due to the richer nature of the object model, the opposite direction (from OQL to SQL) is a bit more complex (or impossible).

Let me now introduce the result of this brainstorming: OQL 1.2. I start with some changes that I like. I go on with some I like less or do not like at all. Finally, I note some semantic differences between the two languages.

### The "good" modifications

consists of a few syntactic changes: some are simple sugaring, whereas others modify more seriously the language.

- In OQL 1.2, one may implicitely construct or access the properties of a structured object. Let me illustrate this with an example that shows that, in some cases, one cannot see the difference between OQL and SQL. The query selects the names and addresses of the three stars hotels.

```
select   name, address
from     Hotels
where    category=3
```

Note that (i) no variable is defined in the **from** clause, (ii) attributes are accessed directly (e.g., *name* instead of *h.name*) and (iii) we find in the **select** clause two subqueries separated by a comma which replace the **struct** subquery required in OQL 1.1. The query result is a bag of tuples with two attributes: *name* and *address*. Of course, one can still use explicit access and tuple construction in OQL 1.2. This new and more concise way to formulate queries is obtained by simply adding syntactic shortcuts, it does not break the functional nature of the language nor its semantics. For instance, consider the above query with a slightly different **select** clause:

```
select   name
from     Hotels
where    category=3
```

The syntax is still SQL. Yet, it returns a bag of strings, not a relation with one attribute as would be the case in SQL. We definitely did not want to change that: it would mean being unable to encompass the full ODMG model.

It is easy to apply some SQL make-up on simple OQL queries but the technique has its limits. For obvious reasons, queries involving accesses to or construction of complex objects/literals will never be SQL-like. For instance, consider the queries of Example 5. Although we can use, for some of them, the above syntactic sugaring, they still remain fundamentally OQL. However, the opposite direction is easier: most SQL queries are OQL-like.

- The independent **sort** and **group** iterators have been removed from the language. In OQL 1.2, the **select-from-where** iterator supports three new clauses: **group by**, **having** and **order by**. I will illustrate grouping later on. The following corresponds to the **sort** query of Example 6:

```
select     c
from       c in Cities
order by   c.countries, count(c.monuments)
```

It does not return a bag, but a list since an **order-by** clause has been used. As in SQL, ascending and descending ordering can be specified. As opposed to SQL, complex queries can be used to specify sorting criterion.

Why do I prefer this new way of sorting collections? Because it supports filtering, sorting and restructuring the elements of a collection using one single iterator (instead of two with the previous syntax). This is very nice and makes me wonder why we did not choose it in the first place.

## The “bad” modifications

are market-driven. They consist in providing two syntaxes (one SQL, one OQL) to the same functionality. You are thinking that I am incoherent: aren't syntactic shortcuts a way of providing two syntaxes? I agree, but this works for shorter and nicer queries. The changes I do not like are those that, in my opinion, do not improve the syntax. They simply allow users to believe they are programming in SQL. I think this is dangerous since, as I explained above and will explain further, some queries cannot be SQL-like. Furthermore, I prefer the OQL syntax. But I should not be fully negative, there is a good thing about these changes: one can ignore them!

The first dual syntax concerns the way we introduce variable or attribute names. Below are two equivalent OQL 1.2 queries, the second one illustrating the added SQL syntax.

```
select   city: name, 3hotels:   select   h
                                from     h in hotels
                                where    h.category=3
from     Cities
```

```
select   name as city,   ( select   h
                           from     hotels as h
                           where    h.category=3) as 3hotels
from     Cities
```

A third formulation removes the **as** keyword in the **from** clause. I aggree, the difference between SQL and OQL syntaxes is slight and prefering one to the other is really a matter of taste. I just find confusing this way of placing the name after the subquery it is attached to, especially when complex queries are involved. Furthermore, there exists a semantic difference between the name attachment in the **from** and **select** clauses. In the former, a variable is introduced to range over the elements of a collection. In the latter, the name is attached to the result of a subquery on one of these elements. I think that it is important to keep the distinction explicit.

As a matter of fact, this semantic difference and the way it is blended in SQL is the reason why I do not like the changes I am going to illustrate now. Consider the following simple SQL query:

```
select   name, price * 5.89 as Price_in_Dollars
from     Hotels
```

Obviously, *price* denotes here the price of one hotel. Now, consider the following one:

```
select   avg(price)
from     Hotels
```

Here, the semantics of *price* has somehow changed: we are applying an **avg** operation on a collection including all hotels prices. Using the usual OQL syntax, this query would be formulated as follows:

```
avg (   select   price
        from     Hotels )
```

thus respecting the functional construction. In OQL 1.2, both syntaxes are now supported. This does not break the semantics of the language since the added SQL-like constructs are viewed as syntactic alternatives. Still, I find it misleading. Indeed, consider the following OQL query which returns the cities' name and number of hotels.

```
select   city: name, nb_hotels: count(hotels)
from     Cities
```

Here, the **count** function is applied in the usual OQL manner, i.e, on the *hotels* set attribute of each city. Although a simple analysis of a query reveals the fashion in which a query uses an aggregate function, I do not think it is a good idea to overload the language in that fashion. Especially since I am convinced that the functional construction is much nicer: the semantics is well defined and, as a consequence, much clearer.

I will refrain from bothering you with more of my personal complaints[†]. All of them can actually be found in the OQL chapter of [31] under the title: Syntactical Abbreviations (note that by removing this section, OQL will be defined in 35 pages!).

### Some Important Semantic Differences:

Traditionally, a **from** clause corresponds to a product between several collections, a product that is usually transformed into a join using part of the **where** clause predicate. In the relational model, the collections are relations and the output of a product/join is a relation whose attributes are obtained by summing those of the input relations (that have to be distinct or renamed). Now, in the object model, things are more complex since we are dealing with collections of many different kinds: e.g., collections of objects, tuples, strings, etc. Thus, the result of a **from** clause is different: it consists of a bag of tuples with one attribute per input collection.

For instance, consider the following query that returns the hotels of Caen with an acceptable price and television in each room:

[†]Interestingly, it seems that some people at the ODMG share my feelings: according to my spy, the debate on OQL 1.2 was heated.

```
select   distinct h
from     h in Hotels,
         f in h.facilities
where    h->check_price() and h.city.name = "Caen" and f = "TV"
```

The output of the **from** clause has the following **bag** type:

```
{{ [h: Hotel, f: String] }}
```

(If the user does not assign variables in the **from** clause, the attribute names are inferred by the system.) Note that this structure is perfectly adapted to the processing of the other clauses which contain operations that have to be applied on each element of the collection in order to filter or build the resulting elements. But it is different from the SQL semantics.

Let us now consider the **group by** clause.

```
select     *
from       h in Hotels
where      h->check_price() and h != Hotel_of_the_Month
group by   city: h.address.city, nb_goodies: count(h.facilities)
```

Note that, as opposed to SQL, a grouping criterion may be more than a simple attribute (which is why we may name them) and that we may use an asterisk in the **select** clause since the model supports nested collections. Now the question arises, what is the resulting type of this query? And the answer is:

```
{ [city: City, nb_goodies: Integer, partition: {{ [h: Hotel] }} }
```

This does look reasonnable: each criterion is represented by one attribute, an added attribute represents the resulting partitions, the type of a partition is that given by the **from** clause. The semantics is as expected: we group all reasonably rated hotels, but the *hotel_of_the_month*, on their city and number of facilities. Thus, to each distinct couple *city, nb_goodies* is associated a collection of hotels of that city and having that number of facilities.

Now, most people would prefer a partition to be a bag of hotels, instead of this bag of tuples. So would I. But, (i) a group can be performed on more than one collection in which case the tuple construct is essential and (ii) I think I made clear that this construct is needed by all other clauses. In the example, we use *h* to compare each hotel to the *hotel_of_the_month*. On top of not being clean, giving two semantics to the same operation misleads the programmers.

Now, let us go further by applying some operations on the resulting partitions. I illustrate this by only giving **select** clauses, the others remain the same.

Counting is simple:

```
select   city, nb_goodies, nb_hotels: count(partition)
```

As expected, the result is of type:

```
{ [city: City, nb_goodies: Integer, nb_hotels: Integer] }
```

Restructuring partitions is also easy:

```
select   city.name, nb_goodies, hotels:   select h.name, h.price
                                          from partition
```

Here, the *hotels* attribute is a bag of tuples with two attributes: *name* and *price*.

Now, I find applying aggregate functions also easy, but this is indeed different from the SQL way:

```
select   city.name, nb_goodies, avg_price: avg(select h.price from partition)
```

All right, it is simpler in SQL. But try to formulate the previous queries in an extended form of SQL (especially the one just before, that requires that one keeps the connection between a grouped hotel name and its price).

## *4.3. OQL Acquires a New Optimizer*

At the end of 1995, OQL 1.2 was the new standard and we had to adapt the $O_2$OQL optimizer/interpreter to this new release. This could have been done simply but I liked the idea of a spring-cleaning and my dear fellows Jean-Claude Mamou and Cassio Souza dos Santos were writting programs that generated ugly queries. Jean-Claude was (and still is) the manager of $O_2$Technology middleware team and used OQL to implement $O_2$Web, a tool with which one can browse an $O_2$ database from the Web (see [6]). Cassio was (and still is) responsible for the ODBC connection to $O_2$. Both these programs generate deeply nested queries that can be unnested.

I felt it was time to, at last, implement some of the great techniques my friend Guido Moerkotte and I developed [37], especially since the OQL 1.2 new **group** operator lended itself nicely to these techniques. Thus, I threw the optimizer code away and built a new one.

Of course, what and how it is implemented is an industrial secret that I am not supposed to divulge. Yet, I cannot resist the temptation to illustrate the fact that researchers do propose techniques that are interesting for the industrial community. Furthermore, it gives me the occasion to silence a silly rumor. I invite people who wrongly think that OQL cannot be optimized to take a look at the literature on the subject. Some pointers to start from: [34, 37, 42]. They illustrate three different approaches of the problem. Also, [44] introduces what I believe to be "the original" object optimization technique: it consists in using joins to evaluate path expressions. This technique has been formalized in different ways, notably in [34, 37, 42]. Let me explain how it works using a simple example.

```
select   h
from     h in Hotels
where    h.address.city.name = "Nice"
```

This query returns the hotels located in Nice. It features a subquery which is a composition of attribute selections: *h.address.city.name*. Such a composition is called a path expression. Suppose that we do not have an index on this specific path expression. One first way to evaluate the query is by navigation: for each hotel in the set *Hotels*, we retrieve the address literal, then the corresponding city object; if its name is "Nice", we add the hotel to the resulting bag. Note that this algorithm checks several times the name of one city object (once for each hotel referencing it). At first glance, this does not seem very costly. Yet, (i) it may entail reading the same city from disk several times and (ii) we can imagine predicates that are more expensive to evaluate (e.g., using a method). Thus, it is important to consider an alternative to the navigation algorithm. The authors of [44] propose a rewriting technique based on joins with class extents. I think the easiest way to understand this rewriting is to consider the following OQL query:

```
select   h
from     h in Hotels,
         c in Cities
where    c.name = "Nice" and h.address.city = c
```

Assume that *Cities* contains all the database city objects. Then, this query is equivalent to the previous one. This is what the technique captures. Extents (in that case *Cities*) are used to break the navigation process into one or more joins. We can now consider a new algorithm: iterate over the cities to extract the identifier of the Nice object; iterate over the hotels and select those whose city sub-component is equal to the Nice object. Here, the rewriting seems particularly appropriate: this is not always the case and depends on the usual factors (physical organisation, selectivity, memory size, etc.).

One last comment: using joins to evaluate path expressions requires extents. As I have explained in Section 2.2, systems implementing persistence by reachability do not maintain them. This does not kill the technique but makes it just more difficult to implement. There are two solutions. The first consists in relying on informations given by users. For instance, a user can state that the names *Cities*, *Countries*, etc. are in fact extents. The other and better solution consists in relying on the system to provide a super-set of each actual extent (which is sufficient but requires caution). This can be done easily since the only difficulty when one relies on a garbage collector is to remove deleted objects at the right time.

## *4.4. OQL Kills its Worst Run-Time Errors*

At the beginning of 1996, another interesting character appeared on the OQL scene: Sophie Gamerman from $O_2$ replaced Guy Ferran at the ODMG and was preparing OQL release 2.0. At that time, she was also Vice President of Support for $O_2$Technology. She is now Product Manager.

Cassio was still in charge of the $O_2$ ODBC connection and worked hard on SQL Null undefined values: they needed a lot of massaging to get into the ODMG picture. Indeed, Null is not part of the ODMG 1.2 model which supports only the Nil undefined object. The reason for this is easy to understand: object-oriented programming languages, which are at the root of the ODMG model, do not support Null literals/values, they only support undefined objects (e.g., Null pointers in C++). Thus, the Nil object and no undefined literals.

But the ODBC connection was not the only provider of problems. Some customers were getting grey hair fighting against OQL's worst run-time error: "access to the Nil object". As I said before, due to the declarative nature of the language, run-time errors are impossible to avoid entirely. Most are not troublesome: I have been programming for some time now and I never divided anything by zero. But, I shamefully admit that I did sometimes try to access a Null pointer in C++ or a Nil object in OQL.

It was time to take drastic measures.

Now, I may be stating the obvious but let me explain why Nil generates run-time errors. Suppose that a Nil object is contained in the *Hotels* set and consider the following query:

```
select   h.price
from     Hotels h
```

Note that I use the SQL syntax, just to give you the whole flavour of OQL. I will come back to my favorite soon.

The query resulting type is a bag of Float. At that time, the Float type did not support an undefined value. So, what can we do with the *h.price* subquery when *h* is Nil? We could forget about it, but that would not be semantically correct and semantics is important as I am sure you agree. Thus, the run-time error. Now, this query is not too bad. One just has to add *h* ! = *Nil* in a **where** clause to remove it. Things get tougher when the Nil error appears in the **where** clause. Suppose Class *Text* has a *contains* method and Consider the following query:

```
select   m
from     m in Monuments
where    m! =nil and ((m.address.city ! = Nil and m.address.city.name = "Paris")
                      or (m.description ! = Nil and m.description->contains("Paris"))
```

A first remark: this is rather cumbersome! Having written this query with great caution, the programmer feels that he/she is on the safe side. Alas, this is not the case. To optimize, an optimizer needs to understand how to move operations, and notably predicates. For this, it usually puts predicates into a normal form, either conjunctive (cnf) or disjunctive (dnf). Both are equally problematic, so let me illustrate my meaning with a cnf. The cnf corresponding to the above predicate is the following:

```
m != nil and
(m.address.city != nil or m.description != nil) and
(m.address.city != nil or m.description->contains(''Paris'')) and
(m.address.city.name = ''Paris'' or m.description != nil) and
(m.address.city.name = ''Paris'' or m.description->contains(''Paris''))
```

From there, the optimizer can do several things one of which is nothing. This may not be the best solution but it does not change the problem and allows me to go straight to the point. So, the first *Monument* object gets in and let us assume that: (i) it is not Nil, (ii) it has a description containing the word "paris" but (iii) it has an undefined city[†]. Accordingly, it passes the first term

[†]Note that it cannot have an undefined address since it is a literal. If the user did not specify this attribute, it has a default value: *street* is an empty string and *city* is Nil.

of the conjunction, the second (since $m.description != Nil$) and third but it generates a run-time error on the fourth. All right, the optimizer has not been smart and should have swapped the two disjuncts of this term. Suppose it does, then the run-time error occurs on the last term. Again, we could swap. But that will just move the problem to Parisian monuments with an undefined description.

A first idea was to introduce **cand** and **cor** operators: their semantics is that of **and** and **or** but they cannot be moved or put in normal form. This meant losing declarativity and forbid many interesting optimization. We did not like that. This is where Cassio's SQL Null came to the rescue and we borrowed once more from SQL.

Remember that in 1996, ODMG did not have Null literals. They appeared later, in ODMG 2.0[†]. So, the idea was to introduce them in the language but forbidding their export: i.e., a Null value cannot be part of the result.

Actually, we did not use the word Null, we used instead **Undefined**. I guess we thought its semantics was different but, in 1998, I do not really see the difference anymore. Anyway, we added two new boolean operations (**is_undefined**, **is_defined**) and defined some rules that are rather simple but very effective. Cassio and the customers are now very happy. The rules are the following:

- The '.' and '->' operations applied to Nil or Undefined returns Undefined.
- All comparison operations (=, !=, <, >, <=, >=) applied with one or two Undefined operands return **False**.
- The **is_undefined** (resp. **is_defined**) operation returns **true** (resp. **false**) if its parameter is Undefined, **false** (resp. **true**) otherwise.
- All other operations on Undefined return a run-time error.

Let me illustrate that we now safely formulate the above query in the following way:

| | |
|---|---|
| **select** | m |
| **from** | m **in** Monuments |
| **where** | m.address.city.name = "Paris" **or** m.description->contains("Paris") |

Now, if either *m* or *m.address.city* or *m.description* is Nil, the predicate returns **False**. Thus, the query is safe. Of course, due to our last rule, all queries are not safe. For instance, the following one may generate a run-time error.

| | |
|---|---|
| **select** | m.name |
| **from** | m **in** Monuments |
| **where** | **not**(m.name = "Crillon") |

Indeed, if *m* is Nil, the predicate is **true** (**not(false)**). Yet, the query in the **select** clause is Undefined and thus we generate a run-time error. Note that this avoids returning an Undefined as part of the result. If one wants to be on the safe side, the **where** clause has to be rewritten in one of the two following ways:

| | |
|---|---|
| **where** | m.name != "Crillon" |
| **where** | **not**(m.name = "Crillon") and **is_defined**(m.name) |

Note that, as in SQL, *m.name != "Crillon"* is not equivalent to **not***(m.name = "Crillon")*. The former returns **false** if *m.name* is Undefined, whereas the latter returns **true**.

[†]Although I find it hard to see how this is going to be implemented in the C++ binding without loosing in performance and user-friendliness.

## 5. CONCLUSION

My story ends here but OQL continues. The language has been implemented in Poet [12], Computer Associates' new Jasmine [3] and JavaSoft's Java Blend [5]. Furthermore, if my information is correct, Apertus [2] is in the process of doing so. In a context of increased communication between heterogeneous systems, my belief is that OQL, due to its perfect conformance with the ODMG rich type system, is the best candidate for exchanging informations. A first step in that direction has been taken when the Object Management Group (OMG [10]) chose it, along with SQL, as a standard to query distributed objects. My hope now is that the software industry recognises the great potential of this language that combines elegance, expressive power and the ability to be easily and, most importantly, formally extended.

Some people seem to think that, as opposed to SQL, OQL does not have a theoritical background. This is not only unfair but wrong. As a matter of fact, whereas the first SEQUEL [33] could pride itself with a formal definition based on the relational calculus, the various extensions of SQL cannot. Indeed, the relational model does not support features such as grouping, duplicates or order. In some way, OQL can be seen as re-introducing in a clear and formal manner these features in a query language. It has been defined on a rich data model, that also has sound theoritical fundations (let me just cite [17]). Its semantics is clear [21] and its functional nature allows it to be extended in a simple way. Furthermore, the literature abounds on works establishing its formal background (e.g. [17, 34, 37, 42]).

I invite you to compare the OQL standard [32] with the query part of the coming SQL3 [8] and make your own opinion. I will be happy to receive any comments on the matter.

*Acknowledgements* —
During this OQL experience I enjoyed learning from many people, including: Serge Abiteboul, François Bancilhon, Catriel Beeri, Claude Delobel, Guy Ferran, Paris Kanellakis, Sophie Gamerman, Dave Maier, Jean-Claude Mamou, Jim Melton, Jeff Mischkinsky, Guido Moerkotte, Claudia Roncancio, Cassio Souza dos Santos and Fernando Velez. I thank them all deeply and warmly.

## REFERENCES

[1] *http://www-rocq.inria.fr/verso/.*

[2] *http://www.apertus.com/.*

[3] *http://www.cai.com/.*

[4] *http://www.gemstone.com.*

[5] *http://www.javasoft.com/.*

[6] *http://www.o2tech.fr.*

[7] *http://www.objectivity.com/.*

[8] *http://www.objs.com/x3h7/sql3.htm.*

[9] *http://www.odi.com.*

[10] *http://www.omg.org/.*

[11] *http://www.ontos.com/.*

[12] *http://www.poet.com/.*

[13] *http://www.versant.com/.*

[14] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. *Very Large Data Bases Journal*, (4:4) (1995).

[15] S. Abiteboul and N. Bidoit. Non first normal form relations : An algebra allowing data restructuring. *Journal of Computer and System Sciences* (1984).

[16] S. Abiteboul and S. Grumbach. Col: a logic-based language for complex objects. In *Proc. EDBT, Venice, Italia* (1988).

[17] S. Abiteboul and P. Kanellakis. Identity as a query language primitive. In *Proc. SIGMOD, Portland, Oregon* (1989).

[18] A. Alashqur, S. Su, and H. Lam. Oql: a query language for manipulating object-oriented databases. In *Proc. VLDB, Amsterdam, Holland* (1989).

[19] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented system manifesto. In *Proc. DOOD, Kyoto, Japan* (1989).

[20] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. Fad, a powerful and simple database language. In *proc. VLDB, Brighton, England* (1987).

[21] F. Bancilhon, S. Cluet, and C. Delobel. $O_2$sql syntax and semantics. Technical report, GIP Altaïr, INRIA, BP. 105, 78153 Le Chesnay, France (1989).

[22] F. Bancilhon, S. Cluet, and C. Delobel. Query languages for object-oriented database systems: the O2 proposal. In *Proc. DBPL, Salishan Lodge, Oregon* (1989).

[23] F. Bancilhon and S. Khoshafian. A calculus for complex objects. In *Proc. Acm SIGACT-SIGMOD symposium on principles of database systems* (1985).

[24] J. Banerjee, H.T. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou, and H.J. Kim. Data model issues for object-oriented applications. *ACM TOIS* (January 1987).

[25] D. Beech. A foundation for evolution from relational to object databases. In *Proc. EDBT, Venice, Italy* (1988).

[26] C. Beeri and Y. Kornatzky. Algebraic optimization of object-oriented query languages. In *Proc. ICDT, Paris, France* (1990).

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

[28] N. Bidoit. The verso algebra or how to answer queries with fewer joins. *Journal of Computer and System Sciences*, (35), pp 321-364 (1987).

[29] M. Carey, D. DeWitt, and S. Vandenberg. A data model and query language for exodus. In *Proc. SIGMOD, Chicago, Illinois* (1988).

[30] R. Cattell, editor. *The Object Database Standard: ODMG 93.* Morgan Kaufmann, Release 1.1 (1994).

[31] R. Cattell, editor. *The Object Database Standard: ODMG 93.* Morgan Kaufmann, Release 1.2 (1995).

[32] R. Cattell, editor. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann (1997).

[33] D. Chamberlin and R. Boyce. Sequel: A structured english query language. In *Proc. ACM SIGMOD workshop on data description, access and control* (1974).

[34] Mitch Cherniack and S. Zdonik. Rule languages and internal algebras for rule-based optimizers. In *Proc. SIGMOD, Montréal, Québec, Canada* (1996).

[35] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. SIGMOD, San Diego, California*, pp. 383–392 (1992).

[36] S. Cluet, C. Delobel, C. Lécluse, and P. Richard. RELOOP, an Algebra Based Query Language for an Object-Oriented Database System. *Data and Knowledge Engineering*, (5) (1990).

[37] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Proc. DBPL, New-York, USA.*, pp. 226–242 (1993).

[38] E. Codd. A relational model of data for large shared data banks. *Communications ACM*, **13**(6), pp 377-387 (1970).

[39] L. Colby. A recursive algebra and query optimization for nested relations. In *SIGMOD, Portland, Oregon* (1989).

[40] G. Copeland and D. Maier. Making smalltalk a database system. In *Proc. SIGMOD, Boston, Massachusetts* (1984).

[41] O. Deux et al. The story of o2. *IEEE Transaction on Knowledge and Data Engineering*, **2**(1) (1989).

[42] L. Fegaras and D. Maier. Towards an effective calculus for object query languages. In *SIGMOD, San Jose, California*, pp. 47–58, San Jose, CA (1995).

[43] D. Fishman et al. Iris: an object-oriented database management system. *ACM TOIS*, **5**(1), pp 48-69 (January 1986).

[44] P. Jenq, D. Woelk, W. Kim, and W. Lee. Query processing in distributed ORION. In *Proc. EDBT, Venice, Italy* (1990).

[45] A. Kemper and G. Moerkotte. Advanced Query Processing in Object Bases Using Access Support Relations. In *proc. VLDB, Brisbane, Australy* (1990).

[46] W. Kim. A model of queries for object-oriented databases. In *Proc. VLDB, Amsterdam, Holland* (1989).

[47] H. Korth, M. Roth, and A. Siberschatz. Extended algebra and calculus for not 1nf relational databases. *ACM TODS*, **13**(4) (1988).

[48] G. Kuper and M. Vardi. A new approach to database logic. In *proc. 3rd ACM PODS* (1984).

[49] C. Lécluse and P. Richard. Manipulation of structured values in object-oriented databases. In *Proc. DBPL, Salishan Lodge, Oregon* (1989).

[50] C. Lécluse and P. Richard. Modeling complex structures in object-oriented databases. In *Proc. PODS, Philadelphia, Pennsylvania* (1989).

[51] C. Lécluse, P. Richard, and F. Velez. O2, an object-oriented data model. In *Proc. ACM SIGMOD, Chicago, Illinois* (1988).

[52] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an objet-oriented dbms. In *Proc. ACM conference on object-oriented programming systems, languages and applications* (1986).

[53] A. Makinouchi. A consideration on normal form of not-necessarily-normalized relation in the relational model of data. In *Proc. VLDB, Tokyo, Japan* (1977).

[54] J. Melton and A. Simon. *Understanding the New SQL: A complete Guide*. Morgan Kaufmann (1993).

[55] Jim Melton. Accomodating sql3 and odmg. http://www.jcc.com/sql_x3h2_95_161.html.

[56] C. Parent and S.Spaccapietra. An algebra for a general entity-relationship model. Technical report, Centre de recherche en informatique de Dijon, France (1985).

[57] P. Pistor and F. Andersen. Principles for designing a generalized nf2 data model with an sql-type language interface. Technical report, IBM Heidelberg Scientific Center (1986).

[58] M. Roth, H. Korth, and A. Silberschatz. Queries and query processing in object-oriented database systems. *Transactions on Office Information Systems* (1991).

[59] M. A. Roth, H. F. Korth, and D. S. Batory. Sql/nf: a query language for ¬ 1nf relational databases. Technical report, Department of computer science, University of Texas, Austin (1986).

[60] S. Rozen and D. Shasha. Rationale and Design of Bulk. In *Proc. DBPL, Nafplion, Greece* (1991).

[61] G. Shaw and S. Zdonik. An Object-Oriented Query Algebra. In *Proc. DBPL, Salishan Lodge, Oregon* (1989).

[62] D. Straube and T. Özsu. Queries and Query Processing in Object-Oriented Database Systems. Technical report, Department of computing science, university of Alberta, Edmonton, Alberta, Canada (1990).

[63] J. Verso. A database machine based on non 1nf relations. Technical report, INRIA, Rocquencourt, France, rapport de recherche no 523 (1986).