

Kaleidoquery—a flow-based visual language and its evaluation

Norman S. Murray
Centre for Virtual Environments
Business House
University of Salford
Salford M5 4WT, U.K.
email:n.murray@salford.ac.uk

Norman W. Paton, Carole A. Goble
Information Management Group
Department of Computer Science
University of Manchester
Oxford Road
Manchester M13 9PL, U.K.
email:<norm, carole>@cs.man.ac.uk

Joanne Bryce
Department of Psychology
University of Central Lancashire
Preston PR1 2HE, U.K.
email:j.bryce@uclan.ac.uk

Abstract

This paper describes the Kaleidoquery visual query language for object databases and its comparative evaluation with OQL. The design philosophy behind the filter flow nature of Kaleidoquery and each of the language's constructs is described, and examples are given that allow comparisons to be made with OQL. This is followed by a description of an experiment with the Kaleidoquery and OQL languages. Two groups of subjects, programmers and non-programmers, were taught aspects of OQL and Kaleidoquery, and then tested under experimental conditions. Results show that both groups answered significantly more correct queries using certain constructs of the Kaleidoquery language.

1 Introduction

Kaleidoquery is a visual query language for object databases, the main design objectives for which were:

- to build a comprehensive visual query language for an object database,
- with compliance to current standards,
- while alleviating previously identified problems noted in query language evaluations, and
- being easier to learn and use than the current standard query language.

Compliance to standards has been achieved through queries being translated into OQL, with support for most of the functionality of OQL. The design philosophy behind the filter flow nature of Kaleidoquery will be described, along with examples of each of the language's constructs, relating them to OQL, and, where relevant, showing how they tackle identified query language problems. The Kaleidoquery language is described independent of any implementation details; an implementation of Kaleidoquery in a 3D environment is described in [30]. The 3D implementation was designed using the framework of [29]. This paper reports the complete Kaleidoquery language and its underlying design philosophies, along with the comparative evaluation of Kaleidoquery and OQL.

The lack of a generally accepted and widely supported query language probably had a significant effect in slowing the uptake of early commercial object-oriented databases. However, the emergence of the Object Query Language (OQL), which is being standardised by the Object Database Management Group (ODMG)[9] and being supported by a growing number of object database vendors, promises to address this limitation. However, textual query languages present several problems to the database user, including the need to know the database's classes, attributes and relationship structures before writing a query, the problems of semantic and syntactic errors, etc.

Visual query languages attempt to bridge the gap of usability for users, and in this paper we present a visual query language for object databases that depicts the query as a filter flow. Visual queries expressed in Kaleidoquery can be translated into ODMG standard OQL using the algorithm presented in [28]. In this way the language can be utilised in any ODMG compliant database that supports OQL.

With a view to establishing whether or not Kaleidoquery provides usability benefits over textual languages, a comparative evaluation of Kaleidoquery and OQL was undertaken, using a paper and pencil test, to compare the design of each of the languages. Many of the early studies of query interfaces to databases were performed using paper and pencil tests [34]. This method is advocated as it avoids the influence of environmental factors[19] on the evaluation, and can be used to highlight difficulties with the query language, although such facilities as interactive help during query creation cannot be simulated adequately using a paper and pencil test. Although

paper and pencil tests fail to evaluate on-line use, they can be useful in evaluating the language, as opposed to on-line experiments where we are evaluating both the query language *and* the interface. With these combined factors, the results of whether one query language is easier to use than another could be inconclusive. It may be the interface that is detrimental to the usage of the less easy to use query language, rather than features of the query language itself.

This paper is organised as follows. In Section 2 the design philosophy of Kaleidoquery is discussed, followed by a specification of the language through examples of its constructs and the corresponding OQL queries. This is followed by Section 3 describing the form of the experiment and Section 4 discussing the results. Section 5 reviews related work in the area of visual query languages to object databases, and Section 6 presents some conclusions.

2 Language Description

Graph based query languages generally use a representation of the database schema as the starting point for querying the database. The user selects parts of the schema and constructs a query from these by combining them with query operators. A problem with graph based queries is that they depict the database schema using entity-relationship diagrams or some other notation that, while readable for a database designer, exploit an abstract symbolic notation that may at first prove cumbersome to the casual user of a database and require some learning. We wanted to depict the query as the user might visualise it in operation, as information flowing and being filtered or refined. When performing a query we start with a mass of information. To this we apply filters or constraints. Our mass of information then passes through these filters letting through only information that we are interested in. We wanted to depict this flow and refinement of information through the query, which is not clearly depicted in graph based queries.

2.1 Identified Query Language Problems

Before the language was designed, previous evaluations of query languages and interfaces were surveyed to highlight any key areas that needed to be addressed. The problems identified were classified under the three headings of language, OQL language and implementation.

2.1.1 Language

These are problems that relate to the query language itself.

Parenthesis Complexity: Increasing complexity when specifying the order of boolean operators with parentheses as the query grows [2, 16, 17, 26, 33, 39, 40].

and & or: Differences in meaning between the English and boolean logic meaning behind the *and* and *or* operations [5, 15, 17, 21, 26, 42]. For example, in English the user may pose the query:

Find me all the pet owners that have cats and those that have dogs as pets.

If this query were translated directly from English into a query language such as SQL, the constraint clause would take the form:

```
where pet = 'cat' and pet = 'dog'
```

The results of this query would always be an empty set as it is not possible for a pet to be a cat *and* a dog—the constraints of the query should have been *ored* together.

Universal Quantification: Errors in the use of universal quantification [16, 21, 40]. Greenblatt & Waxman[16] and Thomas & Gould [40] attributed this error to the ambiguous nature of the English language in the queries that the test subjects were given. Thomas & Gould [40] also suggested that the problem could be due to insufficient training in the use of quantification.

SUM and COUNT: Confusion between the SUM and COUNT operators [40]. Thomas & Gould [40] suggest that system constraints, feedback and modification of the names of the operators would reduce this error.

GROUP BY in SQL: Users did not recognise the need for the function even though they knew how to use it [33, 34, 36, 44]. Reisner [33] attributes this to the operation of the GROUP BY feature not being explicit in English questions. Reisner suggests that the teaching of the function could be altered to stress the appropriate use of the GROUP BY clause, and by attempting to find a supplementary syntactic form which would more closely approximate the English equivalent. Welty & Stemple [44] suggest altering the GROUP BY clause to the participle GROUPED BY.

2.1.2 OQL Language

As OQL is a relatively new language, there have been no reported studies of the query language highlighting any of its deficiencies. The structure of OQL queries is very similar to that of SQL. Any differences in OQL were reviewed to see if they could pose problems for users. The following were identified as likely sources of usability problems.

Relationship Queries: there are two distinct ways of writing a query over a relationship depending on whether it is a single valued or a collection valued relationship (see Sections 2.9 and 2.10).

Structuring Results: altering the structure of results output by a query can involve complex restructuring of the complete textual query (see Section 2.14).

2.1.3 Implementation

These are problems that could be rectified not by changing the language but through the implementation of the language's interface, e.g. user support as they create a query.

Operator Assistance: This covers the following four identified problems:

- Steep learning curve. Users take a long time to learn the language and with infrequent usage they will forget the form of a query and the structure of the database.
- Common syntactic errors [17, 33, 40, 43]. These studies have identified problems such as omitting quotation marks around data values where required. This was noted in approximately 50% of the queries by Reisner [33]. Other punctuation errors include omitting periods, colons, parentheses, etc. These were noted in approximately 20% of the queries examined by Reisner [33].
- Using words from the English query in place of the correct attribute name, e.g. using *Names* rather than *name*, misspellings of keywords and attributes, and the use of synonyms [33, 36, 40, 43]. This could be solved by a direct manipulation interface where the user selects the attributes that are on display rather than keying in the attributes and keywords.
- Applying numeric operators to non-numeric fields [16]. Greenblatt & Waxman's [16] suggestion for solving this is to give attributes more descriptive names to reduce this error. It could also be solved by displaying the type of the attribute along with the attribute and/or by restricting the user so that the option of applying an operator to an attribute is not available. Doan *et al* [13] suggest colouring the attribute according to its data type.

Join Assistance: This is composed of the following two problems:

- Joins—Reisner [33] noted that when creating joins users were confused by attributes with different names and did not think they could link them. The reverse was also true, in that they attempted to join two columns which had the same attribute name but with different data types. Reisner suggests that the database designer choose the attribute names appropriately. This could also be solved by a visual query interface highlighting the attributes that can be used in a query filter depending on their type.
- Cross-linking in one table, [16]. It was noted that users had even greater problems with self joins than with normal joins.

Visual Schema: The structure of the database classes, attributes and relationships is not readily available to users, [25, 37]. This could be solved with a suitable visualisation of the database schema.

Query History: For queries it was noted in [13, 22] that users tended to create new queries by reworking a previous query and so suggests the inclusion of a query history. With these identified query language problems, we shall now examine the visualisation of classes and extents before moving on to the query constructs.

2.2 The Database Schema and Extents

For the examples in this chapter we use the classes and extents shown in Figure 1. This consists of the *Person* class that has the properties *name*, *age*, *children*, *parents*, *employer* and *salary*, and the *Company* class having the properties *name*, *location* and *employees*. The only visible information on the exterior of the class is the iconic representation of the class and its name. All other information pertaining to the class is accessible when the participant selects the class and then chooses to view its attributes. At this point we are not primarily concerned about how the database schema is depicted, but with the nature of the query visualisation. The visualisation of the database schema can almost be considered independently of the query visualisation, except for the visualisation of the extents and classes, as we shall see in the examples. The ODL (Object Definition Language) for the two classes is:

```
class Person
(  extent People)
{
  attribute string name;
  attribute int age;
  relationship set<Person> children inverse Person::parents;
  relationship list<Person> parents inverse Person::children;
  relationship Company employer inverse Company::employees;
  attribute real salary;
};

class Company
(  extent Companies)
{
  attribute string name;
  attribute string location;
  relationship set<Person> employees inverse Person::employer;
};
```

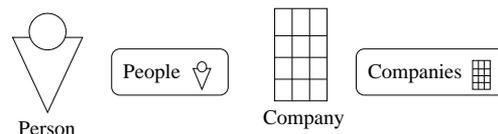


Figure 1: Example Schema

The database extents can consist of all the instances, or a subset of the instances of a particular class. The visualisation of the extents, shown to the right of their respective classes in Figure 1, consists of the extent name and the icon associated with the class of the extent, surrounded by an oval box. In this database there are two extents, one for each class in the schema. The extent allows the participant access to the group of instances, which may be viewed by selecting the extent. As queries are constructed on the schema, the extent group chosen as input to the query will *flow* through the query and be filtered according to the constraints placed on it. Results from a query or a sub query have the new extents, representing the results, placed after the class constraints, and these extents can be used to provide access to the intermediate results or the final results of a query or as input to other sections of the query.

2.3 Results Selection

The simplest form of query allows us to selectively view certain attributes of interest of an instance rather than obtaining the complete instance details. With Kaleidoquery, the initial type of the data will be defined by the extent that flows into the query. It is this type that will be passed along the filter flow pipes, visualised as upward pointing arrows, †, to the results. The type of the filter flow is not altered by the query constraints—the only way

the type of a filter flow can be altered is when it flows into a results box. Along the filter flow it is the instances of the type that are filtered by the constraints. The visualisation of constraints will be examined later.

In Figure 2 we see two versions of the visualisation of the filter flow type where certain attributes of the class *Person* are selected for output, shown along with the corresponding OQL. In Figure 2 (a) the class has the attributes that have been selected for output in the results placed to the right of the class icon. The instances from the *People* extent flow towards the new results extent, but only the attributes shown selected in the results box are allowed to enter the results, here consisting of the *name* and *age* attributes of the *Person* class. This query is the same as the *select* statement of OQL.

```
select tuple(name:p.name, age:p.age)
from p in People
```

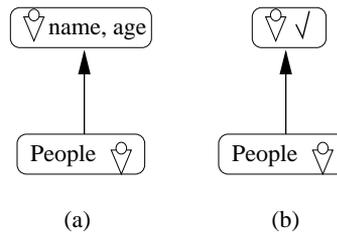


Figure 2: Selecting attributes

If many attributes of the class have been selected to appear in the results, the user may choose not to view them as part of the query but show that some of the attributes have been selected by having a tick placed next to the class, as shown in Figure 2 (b). The attributes that have been selected for inclusion can be examined by selecting the results and choosing to see the result's attributes. If a query is placed on the class and none of the attributes have been selected for inclusion, then the default is to view all the attributes of the class, as can be seen in the query in Figure 3. Towards the end of this section it shall be shown how to structure, order and group the query results.

2.4 Simple Query Constructs

The query shown in Figure 3 includes a simple constraint that restricts the results to include only persons who are aged less than 20. In this case no attributes in the person class have been selected for viewing, so by default the results consist of the group of people satisfying the query. Any of the operators $=$, $>$, $>=$, $<$, $<=$ and *like* can be used in these queries. We shall now look at how simple queries can be combined using boolean expressions.

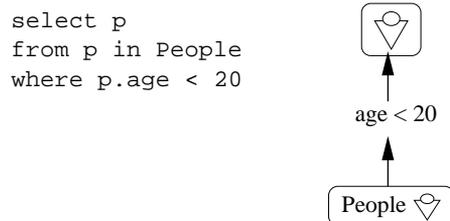


Figure 3: A simple query

2.5 And, Or and Not

When specifying *and* and *or* we can use a representation similar to that used to represent the operation of *and* and *or* gates in electronics, as shown in Figure 4. In this diagram we see the inputs (constraints) of the gate represented as the switches. In the case of a single constraint, as seen in Figure 4 (a), we see the constraint acting as a single switch. If the constraint holds true then the instance will pass through the switch, lighting the bulb. With the *and* gate, Figure 4 (b), the bulb will only light when both switches are closed (true) but with the *or* gate, Figure 4 (c), the bulb will light when either or both switches are closed (true). In the previous example we only had one filter or switch but now we see how multiple filters or switches can be handled in a query using this form of representation.

An early example of this form of filter flow was proposed by Shneiderman [38]. Shneiderman used the metaphor of water flowing through a series of pipes and filters, where each filter would let through only the appropriate items and the layout of the pipes indicated the relationships of *and* and *or*.

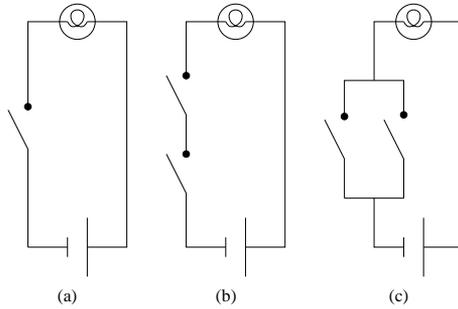


Figure 4: Representing multiple filters using switches

The filter flow model was chosen to visualise the *and* and *or* relations of the query as it has been previously shown that boolean logic has proven difficult for large sections of the population (see Section 2.1). Preliminary results have shown that the filter flow representation is more favourable than a text only SQL interface [45]. How *and* and *or* are specified in a query is now examined. Figure 5 shows visualisations of four queries. Figure 5 (a) shows the *and*-ed visualisation of the OQL query:

```
select p
from p in People
where p.age < 20 and p.name = ``Smith``
```

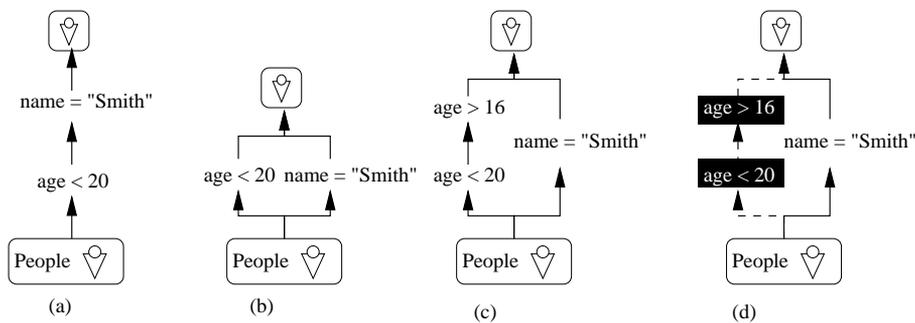


Figure 5: Visualising *and*, *or* and *not*

The instances from the input extent flow through the query and are filtered through each constraint in turn. Figure 5 (b) shows how an *or* is visualised by the extent instances flowing into each part of the *or* query and combining at the top of the query to produce one group of results. The OQL of the query is the same as the above example except that the constraints have been *or*-ed. In Figure 5 (c) we see a more complex query using both *and* and *or* expressions. The OQL for this is as follows:

```
select p
from p in People
where (p.age < 20 and p.age > 16) or (p.name = "Smith")
```

With *not* we can either select an individual constraint or we can select a group of constraints that we wish to apply the negation operation to. If a single constraint is chosen then this constraint is highlighted by darkening or inverse videoing the constraint. If whole branches of the query have been selected for the *not* operation, then in addition to the constraints of that branch being shown in inverse colour, the flows linking the constraints can be highlighted also, in this case shown as dashed lines, see Figure 5 (d).

2.6 Joins

In Figure 6 (a) we see a query utilising more than one extent and comparing attributes of the class of each extent. Here we have introduced another extent called *NewPeople*. The query obtains the names of the people in the extent *People* that have the same salary as people in *NewPeople*. The queries shown in Figure 6 can be seen as

having two sets of results as output. These are the results from each of the data flows and each of these represents a valid OQL query. The equivalent OQL query to the Kaleidoquery will retrieve the results set that is positioned highest in the figures. In Section 2.14 we shall see how we can combine data flows to structure the results.

- (a) `select p.name
from p in People, q in NewPeople
where p.salary = q.salary`
- (b) `select p.name
from p in People, q in People
where p.salary = q.salary and p != q`

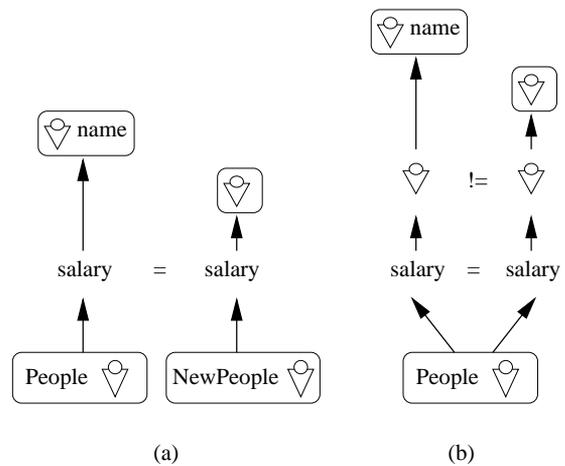


Figure 6: A join and a self join

A self join can be performed by using the same extent as input to the two flows, as seen in Figure 6 (b). In this query, it is also shown how an instance of the data flow (rather than just specific attributes of the instance) can be utilised in a filter. In Figure 6 (b), the query obtains all distinct people with the same salaries.

2.7 Aggregates

Aggregate functions, such as count, sum, avg, etc, can be applied to collection results or collection extents as shown in Figure 7. The extent to be aggregated is selected and then the aggregate function is applied to the extent. The aggregate function is then surrounded by another results box. The query in Figure 7 computes the number of persons in the extent People. Aggregates can also be applied to derived collections and collection attributes of a class. For example, Figure 15 finds the maximum salary in a set of people's salaries.

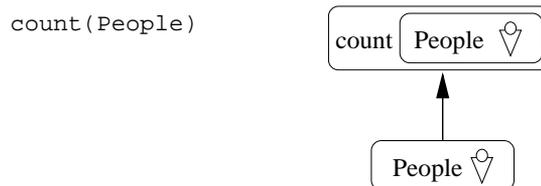


Figure 7: Aggregate function evaluating the size of the extent People

2.8 Arithmetic

Figure 8 shows an arithmetic expression. The participant can select attributes in a class and apply arithmetic operators to them. In Figure 8 the attribute weight of person has been multiplied by 2.205 to change it from kg to lbs. The participant also has the option of naming their expression, and, as can be seen in Figure 8 (b), the new attribute lbs has been selected for inclusion in the results.

- (a) `select (p.weight*2.205)`
`from p in People`
- (b) `select struct(lbs:p.weight*2.205)`
`from p in People`

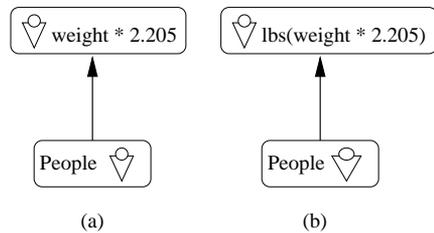


Figure 8: Arithmetic

2.9 Path Expressions—single valued relationships

In Figure 9 we see how a query can be visualised that has been formed through navigation from one class to another related class. To place a constraint on a person’s employer, the user has to navigate from the Person class to the Company class. The navigation along the relationship is visualised as a horizontal arrow with the relationship name located above the arrow. After the arrow the related class is displayed. From the Company class we can select the attribute *name* and place the constraint that it be equal to “ADL”.

```
select p
from p in People,
where p.employer.name = "ADL"
```

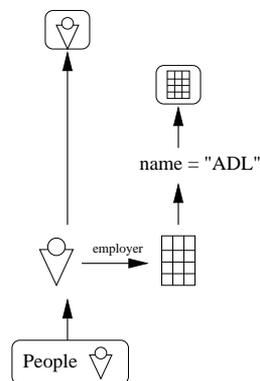


Figure 9: Path expressions

2.10 Path Expressions—collection valued relationships

In Figure 9 we only added a single constraint to the related class’s attribute. In Figure 10 we see how multiple constraints over related attributes are visualised in the same way as was done previously in Section 2.5 on boolean operators. We have navigated from the *Company* class to the set of *employees* of that company. As before, this relationship is visualised as an arrow going to the right with the relation name placed above the attribute. As this is a collection valued relationship, in that many employees work for a company, the Person icon is surrounded by an oval box to show that there are a group of Persons rather than a single entity. To the members of this set we apply constraints to find the employees that are aged over 60 *or* who have salaries of equal or greater value than 25000. The collection of people that satisfy this sub-query is visualised above these constraints. This collection can be viewed separately from the main query of retrieving the company names and can also be used to create structured results (see Section 2.14). From this example we can see how queries using *and* or combinations of *and* and *or* are handled.

For collection valued relationships we see that, as the relationship is over a set, the set to be iterated over in the navigation must be specified in the *from* clause—*e in c.employees*. The variable *e* is then used in the *where* clause for obtaining the employees aged over 60 or that earn 25000 or more. This is in contrast to the single valued relationships described in the previous section where for a constraint, navigation over the relationship was only specified in the *where* clause. In the Kaleidoquery language the only difference between the queries is shown by the result of the collection valued relationship being enclosed by an oval box to indicate that the relationship yields a set.

```

select c.name
from c in Companies, e in c.employees
where e.age>60 or e.salary>=25000

```

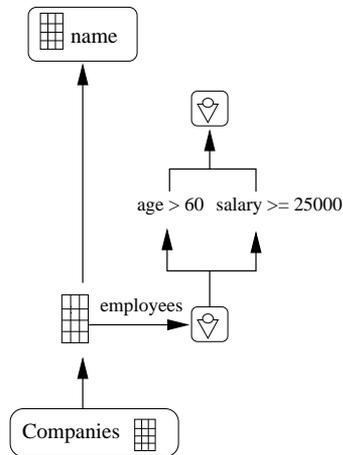


Figure 10: Path expression utilising *or*

In Kaleidoquery we have removed the need for the user to learn two different ways of writing navigational queries that in OQL are dependent on the cardinality of the relationship. With Kaleidoquery the only difference between the two relationship queries is that an oval set box is placed around the class icon of a collection valued relationship.

2.11 Membership Testing and Quantification

For membership testing and universal and existential quantification we have introduced a new edge, \curvearrowright , that links the operator with its two operands. As subqueries are evaluated, the results of these queries can be used as input to other queries. Figure 11 shows this in action. The query finds all employees that work for companies located in England. A membership test is performed on the results of the query “what companies are located in England”.

```

select p
from p in People
where p.employer in
(select c
from c in Companies
where c.location = "England")

```

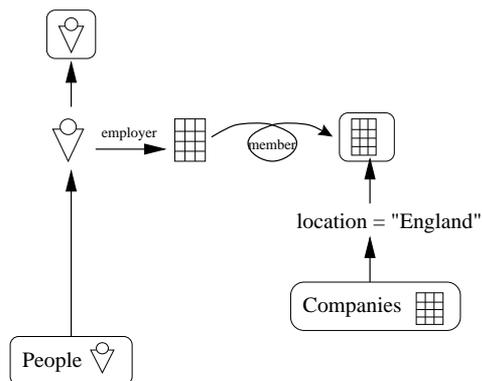


Figure 11: Membership testing

Figure 12 depicts two queries using the *for all* and *exists* operators of OQL. These queries are very similar to the one shown in Figure 10, except that the filter flow arrow that leaves the employee set is annotated with the *all* or *exists* operation. To highlight that these functions have been applied to the filter flows we could alter the colour of the filter flow. The OQL queries that are equivalent to Figures 12(a) and 12(b) respectively are as follows:

(a)

```

select c.name
from c in Companies
where for all e in c.employees:
    e.age>60 or e.salary>=25000

```

(b)

```

select c.name
from c in Companies
where exists e in c.employees:
    e.age>60 or e.salary>=25000

```

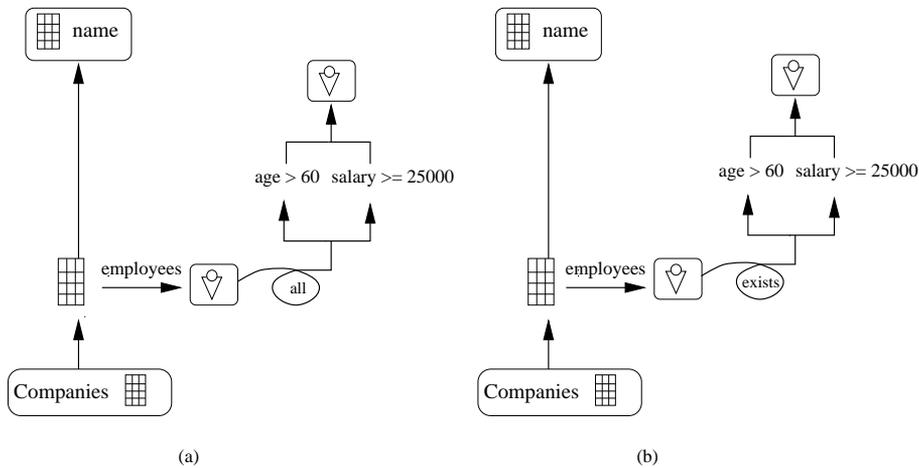


Figure 12: Universal and existential quantification

The queries shown in Figure 12 do not depict the flow of information as naturally as has been seen in previous queries. The operations *all* and *exists* have been applied to the flow and as such are not a restriction on the flow but an overall operation on the flow checking that the collection of instances in the flow satisfy the stated restrictions.

2.12 Binary Set Expressions

The standard set operators are defined on collections of type set and bag in OQL are *union*, *intersect* (intersection) and *except* (difference). These operations are shown as converging data-flows annotated with an icon representing the binary set expression. The union expression could be shown simply as a combination of the data-flows but this is avoided so as to have three consistent icons for the expressions. The three icons can be seen in Figure 13. Each icon depicts a cross-section of the data-flows with the sections of the data-flow that are to continue shown shaded.

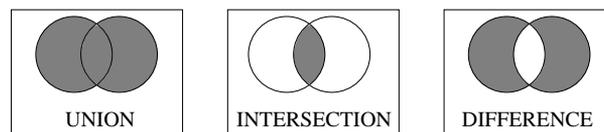


Figure 13: Set expressions

In Figure 14 we see an example of the intersection set expression in a query, where the intersection of people that work for IBM with those that work in London.

2.13 Combining operators

A more complex query is shown in Figure 15 that uses a combination of the operators that have been introduced so far. This query finds the names of companies that have employees older than 60 or that receive salaries greater than the maximum salary obtained by the Smiths that work for companies located in England. In this query we combine related classes, *and* and *or*, aggregates, and membership testing to create the visual query.

2.14 Structuring Results

One of the problems with writing an OQL query is that the way in which a query is structured influences the ways in which the result can be structured. In Kaleidoquery we have tried to remove this complication by separating the two tasks. The participant first builds their query, with the results being visualised above the query. The participant can then select, from the available options, how the results are to be structured. When satisfied they can then view the results by examining the output.

With the queries in Figure 6 we may wish to view the results as pairs of related people with the same salary or as individuals listed with a group of people with the same salary. In OQL this is possible with the operators *struct* or *tuple* (both of which have the same behaviour) and correct nesting of the query. The OQL and the Kaleidoquery visual equivalent for obtaining pairs of related people is shown in Figure 16 (a). Here we see that

```

(select p
 from p in People
 where p.employer.name = ``IBM'`)
intersect
(select q
 from q in People
 where q.employer.location = ``London'`)

```

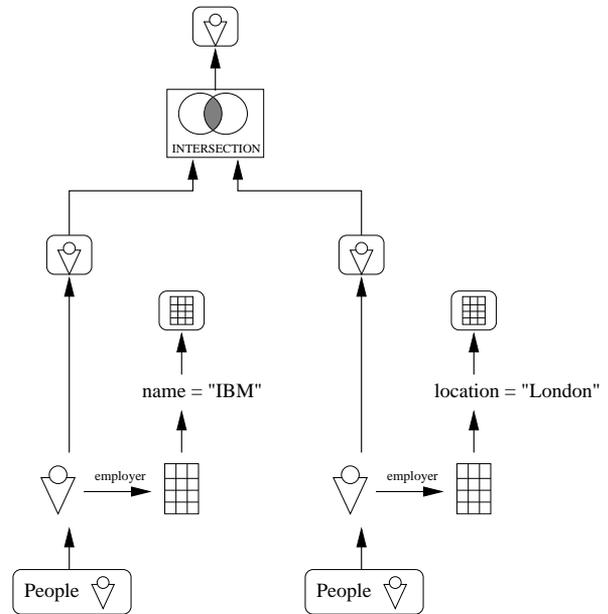


Figure 14: Set expressions example

```

select c.name
from c in Companies, e in c.employees
where e.age>60 or e.salary>=
max(select p.salary
 from p in People
 where p.name = "Smith"
 and p.employer in
 (select c
 from c in Companies
 where c.location = "England"))))

```

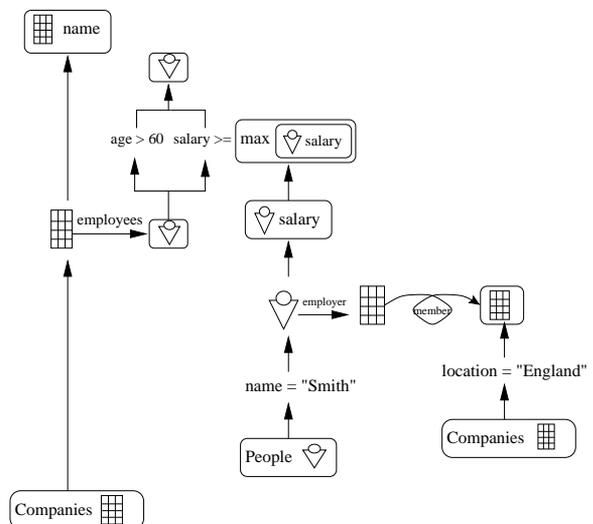


Figure 15: Complex Query

we wish to link each instance on the left of the query with each corresponding instance on the right where the two instances have equivalent salaries, i.e. we want to link them on a *per item* basis. This operation has been visualised with a structure linkage arrow, $\overline{\rightarrow}$, with the style of structuring that we require, in this case *item*, placed over the structure linkage arrow. At the top of Figure 16 we see a visualisation of the final structure of the results. Figure 16 (b) presents the OQL and the Kaleidoquery for generating people with groups of people with equivalent salaries.

Comparing the OQL queries, we can see how different in nature they are. If we wish to obtain nested results then we can see that in OQL the queries require to be nested. With Kaleidoquery we can see that to change between the two queries is simply a matter of changing the structure operation; no other changes need to be made to the query as we have separated the tasks of writing the query and structuring the results.

```
(a) select tuple(pname:p.name, qname:q.name)
      from p in People, q in People
      where p.salary = q.salary
```

```
(b) select struct(p.name,
                  (select struct (q.name)
                   from q in People
                   where p.salary = q.salary))
      from p in People
```

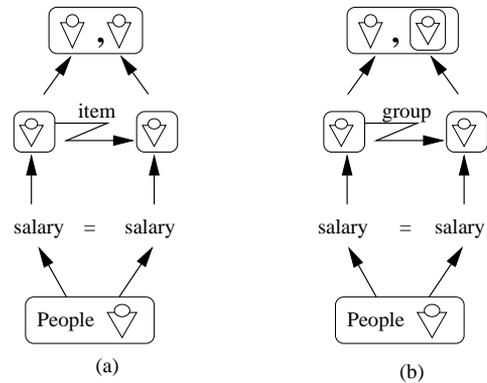


Figure 16: Structuring the results

2.15 Ordering the results

In OQL, the results of a query can be sorted via the OQL operation *order by*. Visually, this can be done by annotating the results with an order. If the results are nested then the ordering inside the nested results is independent of the ordering outside the results. In Figure 17 we see how the sort operator has been applied to the results of a query obtaining the name and location of companies with a list of their employee’s names and ages. The results of the query can then be annotated with numbers showing the order in which they are to be sorted. Along with this number is the ordering associated with the attribute. At present in OQL, results can be ordered in ascending, *asc*, or descending, *desc* order. These are visualised by an upward pointing arrow, \uparrow , for ascending and a downward pointing arrow, \downarrow , for descending. An advantage of this method of sorting over OQL is that the user can perform all the sorting at the final stage. With OQL, nested results need to be sorted before they are used as part of the results, as can be seen in the OQL for the query in Figure 17.

```
select struct(name:c.name, location:c.location,
             employee:(
               select struct(name:p.name, age:p.age)
                 from p in c.employees)
             order by employee.age desc
           )
from c in Companies
order by company.name asc, company.location asc
```

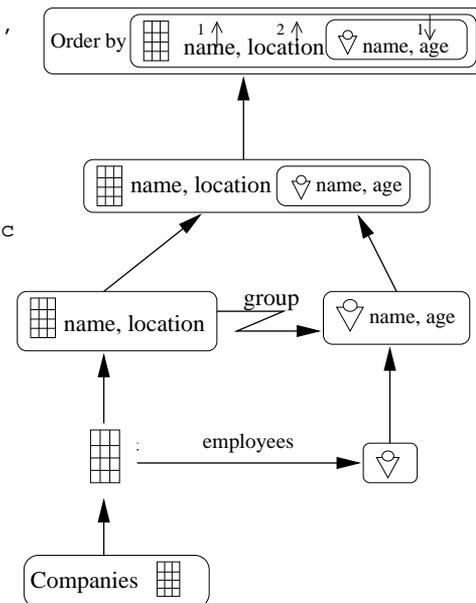


Figure 17: Ordering the results

2.16 Grouping Results

The results of a query can be partitioned into groups by application of conditions on the results. The result set is then split into subsets that satisfy the relevant partitioning conditions. Results can overlap between partitions, and

there can also be results that satisfy none of the partition queries.

Figure 18 shows a query where the name, age and salary of company employees are selected. We can then apply the *group by* operation to the results. This allows us to separate the results into distinct groups depending on the conditions that they satisfy. The two partitionings cause the results set to become a set of results sets, as seen at the top of Figure 18.

```
select struct
  (name:p.name, age:p.age, salary:p.salary)
from p in People
group by
  part1:x.name = "Smith",
  part2:x.salary>=30000 and x.salary < 40000
```

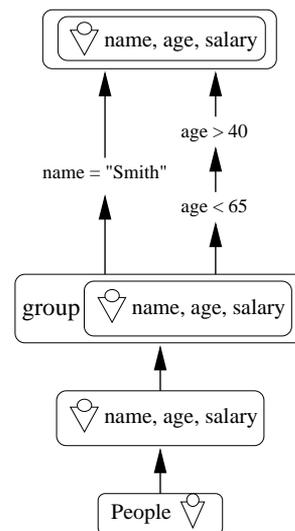


Figure 18: Grouping the results

After the results have been converted into partitions, a *having* clause can be used to filter the results using aggregation functions that operate on each of the partitions. In the following query (the same as the above except for the last line), the *having* clause calculates the average age of each of the partitions and only where this value is less than 50 will the partitions be present in the results. The visualisation of the *having* clause of the query can be seen at the top of Figure 19.

2.17 Omissions from OQL

Certain aspects of OQL have not been implemented in Kaleidoquery and each of these will be described in turn. They cover the areas of constructors, accessors, and conversion expressions.

Constructors Kaleidoquery does include the equivalent of `struct` for constructing the results of a query, but it does not include the equivalents of `set`, `bag`, `list` and `array`. These operations create collections of the operation name type, from individual elements of the same type. For example, `set(1, 2, 3)` returns the set of the three elements 1, 2, and 3.

Accessors Kaleidoquery contains the accessors for obtaining attributes of classes and related classes, but does not include method calls and accessors for lists and arrays, e.g., `first` and `last` to access elements of a list, and indexing for arrays and lists.

Conversion These cover the OQL expressions `listtoset` for converting a list to a set; `flatten` for collapsing a set of collections into a single collection; and `element` that returns the single element in a collection (where the element is a singleton).

```

select struct
  (name:p.name, age:p.age, salary:p.salary)
from p in People
group by
  part1:x.name = "Smith",
  part2:x.salary>=30000 and x.salary < 40000
having avg_salary:
  avg(select x.salary
    from x in partition) > 50000

```

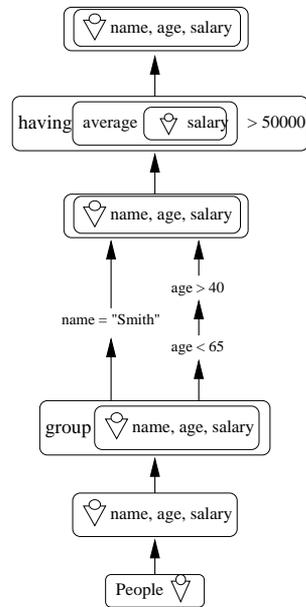


Figure 19: Filtering partitions

3 Evaluation

In the previous section, the Kaleidoquery language was described. In the next two sections we present an evaluation that was performed on Kaleidoquery.

Before we can perform our evaluation we need to decide which form of evaluation strategy to take. The type of evaluation strategy taken normally dictates the form of results that will be gained. For example, whether we are trying to improve the usability of the language—that could be achieved through some form of collaborative or think aloud protocol, or whether we are trying to test if the language meets some predetermined criteria or to test certain hypotheses.

The evaluation performed was a paper and pencil experiment to compare the Kaleidoquery and OQL languages. The aim of the evaluation was to obtain quantitative data on user performance and to attempt to prove or disprove certain predefined hypotheses. Qualitative data was also obtained with user questionnaires. It was decided to compare Kaleidoquery with OQL to assess whether it could be proven that certain areas of OQL such as relationships and structuring that were identified in Section 2.1 could be improved over in a graphical query language. Furthermore, the authors know of only one other evaluation of OQL, though this evaluation did not highlight any areas where the languages were better than each other[11].

The aim of this evaluation is to compare two query languages and assess whether one is easier to learn, understand and use than the other. To do this it is necessary to:

1. Identify the types of tasks the user will perform, as described in Section 3.1.
2. Define what we are going to measure, e.g., ease of use, user satisfaction, etc., and how we are going to measure it, e.g., user scores, questionnaires, as described in Section 3.2.
3. Perform the evaluations and measure the relevant parameters of user performance, as described in Section 3.3.
4. Analyse the results, as described in Section 4.

3.1 Relevant Query Tasks

Table 1 from Reisner [35] lists some of the tasks that have been used for measuring ease of use in query languages. The task of query writing has been chosen for use in this evaluation as from this we can obtain the most data about how easy it is to learn and use the language, and because it is the most commonly used task type in query language evaluations.

Task	Description
Query writing	Users are given a question stated in natural language and required to write a query in the given query language.
Query reading	Users are given a query written in the query language and asked to write a translation into natural language.
Query interpretation	Users are given a query in the query language and a printed database with the data filled in. They are asked to find the data asked for by the query.
Question comprehension	Users are given an natural language question and a printed database and are asked to find the data asked for.
Memorisation	Users are asked to memorise and reproduce a database.
Problem solving	Users are given a problem and a database and are asked to generate questions in English that would solve the problem. The questions should be answerable from the database.

Table 1: Tasks Used to Measure Ease of Use in Query Language Evaluation, from [35]

3.2 Data Collection

Having chosen to use query writing in our evaluation, the hypotheses to be tested need to be specified. The experiment is then structured to test for these hypotheses. The main hypotheses cover:

- ease of use—measured by scoring the subjects queries, measuring the number of errors made, and analysing the types of errors made.
- user satisfaction—obtained as feedback from the subjects during the evaluation via questionnaires. These can find user’s likes and dislikes, and support feature ranking where users are asked to rate each of the query language’s features.
- resolution of query language problems—from the scores obtained from the queries that subjects attempt, we can compare query languages features to see whether certain components of the language are easier to use than others. In Section 2.1 query language problems were identified. In this evaluation we compare the query languages and identify whether there is any preference in the way features listed in Table 2 are supported. With these identified query language features we can generate appropriate tasks for the user to attempt.

Query Feature
1. selecting attributes to be shown as the results of a query;
2. creating queries comparing attributes;
3. parenthesis complexity and the use of <i>and</i> and <i>or</i> versus the data flow method in Kaleidoquery;
4. path expressions;
5. structuring the results.

Table 2: Query language features to be evaluated

3.3 The Experiment

The rationale for the experiment is to evaluate the OQL and Kaleidoquery languages via query writing, with query scores and user questionnaires used for data collection. Within the experiment there are sub-tests that allow the comparison of key query language features. Details of the subjects assigned to the evaluation and the form of the experiment follow.

3.3.1 The Subjects

Two subject groups have been chosen for the experiment: programmers and non-programmers. Programmers are those familiar with computers, programming languages, databases and query languages such as SQL, e.g. , people with a background in computer science; and non-programmers, those that are familiar with computers but who have no programming language experience, and have not used a query language such as SQL before, although they would probably have used other forms of database interface such as those used in libraries.

In examining the ease of use of OQL and Kaleidoquery it could have been deemed necessary to use only subjects that have never used SQL before due to the similarities of SQL with OQL, to reduce transfer effects. These transfer effects could lead to a bias in the scores of the subjects using OQL, as they are familiar with SQL. This could lead to increased scores due to the familiarity with SQL.

In our experiment we chose to use two groups of subjects so that we could analyse how well non-programmers coped with query languages that they had never come across before; how programmers managed with textual query languages, that they tended to be more familiar with, against a visual query language; and also allow for comparison between the two groups to see if any significant differences are highlighted.

3.3.2 Form of the Experiment

The number of subjects required for each of the groups is 10. In undertaking an evaluation we need to obtain reliable results as well as keeping costs to a minimum, both in time and expense, by using the minimum number of subjects. Johnson [20] states that six subjects is the minimum number required for a controlled experiment, but that the increase in sensitivity with ten or more subjects is considerable. With this in mind we will aim to evaluate 10 users in each group, giving a total of 20 subjects. All the subjects used have a university degree.

As the subject groups will be evaluated using both query languages it will be a within-groups experiment, in each of the individual groups, and a between groups experiment comparing each of the subject groups. To reduce bias, each of the groups was split into two to counter balance the influence the order of the presentation of the 2 query languages. This is performed by alternating the subject sequences between the 2 query languages, so that any confounding variables that appear during the experiment will have a high chance of affecting the Kaleidoquery and OQL languages equally (and therefore cancelling out in the subsequent analysis). Within-group evaluation can improve precision and increase the efficiency of an evaluation, but there is an increased risk of transfer effects, resulting in the need for some careful analysis of the results to check for these effects.

Therefore it was required to have four evaluation sessions with five subjects in each: two sessions for each of the two types of groups, combined with the need to do two sessions with each of the groups to alleviate the transfer effects.

A test session with a non-programmer (and also a psychologist) was performed so that the evaluation technique could be verified, and teaching materials and questionnaires tested, to see that sufficient data is collected for analysis and to save having to redo the evaluations due to unforeseen difficulties.

3.3.3 Training

It was not possible to teach the complete query languages to the subjects. Only the five previously identified areas (see Table 2) of the query languages were taught to test for the hypothesis regarding each of these areas. During the training each of these areas was introduced to the subjects by an example and explanation. The subjects then attempted a query that utilised the query feature. The subjects also gave themselves a mark according to how correct they thought their answer was. This mark was used to help identify subjects that may be having difficulties and also to give them an introduction to self marking, as the subjects would be required to self mark during the testing phase of the evaluation. Any difficulties the subjects experienced were dealt with and the answer to the query was given. The training composed of 7 example queries where a topic was introduced with an example query for them to tackle. Training sessions for each language lasted approximately one hour.

A simple example database was devised that was sufficient for expressing the queries that we are interested in without being so large as to overwhelm the user with classes and attributes. This was composed of an employee, company and orders database.

3.3.4 Evaluation Queries

After the language training phase, subject performance was evaluated through query writing. Each subject was given a worksheet with five English queries that were to be rewritten. The subjects self assessment their attempts at the queries were recorded on the sheets. The queries used were relatively simple as the subjects had just been introduced to the language and if the queries were too complex then they may have been unable to answer any of the questions. The same queries were used for both groups so a comparison could be made between them. The subjects answered five queries in the evaluation sessions (see Appendix A). This took them on average about 20 minutes.

3.3.5 Questionnaires

The questionnaires attempted to gauge the users satisfaction with each of the query languages. After the assessments on each query language, subjects completed a questionnaire, with the exception that after completion of the assessment on the second query language the questionnaire contained supplementary questions asking the subjects to compare aspects of the two languages. The results of the questionnaire can be split into four sections:

1. Overall reactions—queries to obtain an overall reaction to one of the query languages.
2. Query language constructs—subjects rate the ease of use of specific aspects of one of the query languages.
3. Query language comparisons—subjects are asked to compare specific aspects of both query languages and rate any preferences they have.
4. Subjects comments—allows the subjects to comment freely on the query languages.

For the first two sections of the questionnaire the subjects rated the query languages independently. These ratings were then compared to see if there were any significant differences between the values given by the subjects. The questionnaires and training and evaluation queries can be found in [28]. The evaluation queries are presented in Appendix A.

4 Results Analysis

Grading of the tasks was assessed using the error classification scheme first proposed by Reisner [36], ordered by increasing severity of error:

- C = completely correct.
- D = minor data error, the data is not supplied completely as required, e.g. only the surname is supplied when the full name is required.
- M = minor language error, e.g. misspellings and punctuation.
- S = error of substance; these are valid queries that would run but produce the wrong answer.
- F = error of form, for invalid queries.
- N = query not attempted.

For statistical analysis of correct responses, scores, etc., the t -test was used [32]. The t -test is derived directly from the standard method used to estimate significance of a deviation from the mean in a normal distribution, and is intended for use where the number of samples is small. Two styles of t -test were used:

- paired sample t -test—these were used to compare the two languages within the same test group.
- independent samples t -test—these were used to compare a single aspect of one of the languages between the two subject groups.

From the t -tests we obtain t and p values. The t statistic measures the difference between the two variables' means, with the p value defining the significance of the t statistic. Values of $p \leq 0.05$ indicate a significant result. If $p = 0.05$, this indicates that there are only 5 chances in 100 that the result was obtained due to chance.

4.1 Analysis of Evaluation Queries

4.1.1 Overall Evaluation and Comparison

For each of the subject groups, the correct responses (C) were combined with those containing minor language (M) and minor data (D) errors to obtain the mean percentage of *essentially correct* responses. These results are shown in Table 3. These results show that:

- The results of a paired value t -test indicate that non-programmers answered significantly more correct queries in Kaleidoquery than in OQL (t 4.63 : $p = 0.001$).

- Programmers scored better than non-programmers in each of the tests. Independent sample t -tests gave the following results: OQL (t 8.55 : p = 0.000); Kaleidoquery (t 3.73 : p = 0.002); combined (t 7.78 : p = 0.002). It can also be seen that the significant difference is larger in the case of OQL.
- There was no significant difference between the scores programmers obtained for Kaleidoquery and OQL.

Response Category	Non-programmers		Programmers	
	OQL	Kaleidoquery	OQL	Kaleidoquery
C (Correct)	4	30	58	48
D (Minor data error)	6	4	2	12
M (Minor language error)	0	12	14	22
Total <i>essentially correct</i>	10	46	74	82
S (Error of substance)	4	16	0	2
F (Error of form)	68	38	26	16
N (Not attempted)	18	0	0	0
Total incorrect	90	54	26	18

Table 3: Percentage of query responses in each category

From Table 3 it can be seen that the subjects performed more errors of form with OQL than Kaleidoquery and also that the subjects performed more errors of substance in Kaleidoquery. Although these results are not significant, they are an indication that when users created their queries wrongly with Kaleidoquery, the queries would still be executable. With OQL the subjects created many more errors of form rendering their queries unexecutable.

4.1.2 Analysis of Errors

Table 3 breaks down the responses into the six categories for the four populations in the experiment. It should be noted that the M and D scores do not represent the total occurrence of minor language and data errors in the experiment, because each response is placed in the category of the worst error it contains.

Table 4 lists the features that were tested in the evaluation (see Section 3.2) and identifies whether there was any significant difference between query responses obtained from the subjects. If there is a K in the table, then subjects answered significantly more correct queries in Kaleidoquery than in OQL. An O indicates that subjects answered significantly more correct queries in OQL than in Kaleidoquery. A blank indicates that there was no significant difference in the responses from the subjects.

Language factor	Programmers	Non-programmers
1. Selecting attributes		K (t 3.99 : p = 0.003)
2. Attribute comparison		K (t 3.67 : p = 0.005)
3. Data flow		
4. Relationships	K (t 3.86 : p = 0.004)	K (t 4.31 : p = 0.002)
5. Structuring		

Table 4: Language factors found significantly better when comparing Kaleidoquery and OQL

Table 4 shows that the non-programmers answered significantly more correct queries using selection of attributes, attribute comparison and relationships in Kaleidoquery. The programmers answered significantly more correct queries using relationships. Each of the subject groups answered more *essentially correct* queries in Kaleidoquery than in OQL. The errors made by each of the subjects groups will now be examined.

Programmers errors: The majority of errors by programmers in OQL was with queries over relationships (as has been highlighted by the statistics in Table 4). The larger number of completely correct queries formulated in OQL could be attributed to programmers familiarity with programming languages. In Kaleidoquery the programmers made no statistically significant errors in any one area, with errors distributed over each of the query language constructs.

Non-programmers errors: Non-programmers made significantly fewer errors in three out of the five tested areas of Kaleidoquery. The two areas without any significant difference were:

- **Data flows:** A previous evaluation by Young and Shneiderman [45] specifically compared data flows with *and* and *or*, with results favouring the data flow method. The reason for these results not being repeated could be attributed to the simple nature of the queries in our evaluation as regards boolean expressions. The queries in the evaluation contained no more than two boolean expressions, and so did not thoroughly test the limitations of the data flow or the textual equivalent.
- **Structuring:** Subjects did answer more correct queries using structuring in Kaleidoquery than in OQL although this difference was not significant. Examining the queries shows that users attempted all the queries with structuring in Kaleidoquery, but attempted less than two thirds of the structuring queries in OQL. This could indicate that they found the separated tasks of querying and then structuring easier in Kaleidoquery, than in OQL where the tasks are, in effect, combined. A problem noted with Kaleidoquery was that users omitted the structuring arrow, even though they structured the results at the top of the query correctly (such queries were considered incorrect). Another problem was a tendency to completely omit result structuring. Queries ended with two results collections at the top of the query and it could have been assumed that this was the correct output of the query, whereas that results needed to be structured together to form one output. This helped contribute to the amount of errors of substance that are seen in Table 3 by the non-programmers and this aspect of the query language will require further evaluation to ascertain its value in Kaleidoquery.

4.2 Self Assessment

After subjects attempted each of the questions in the evaluation, they rated their attempt in the range 0–5, with 5 being a completely correct answer and 0 a completely wrong answer. For programmers, there was no significant difference, but for non-programmers, the results of a paired sample *t*-test indicate that the subjects gave significantly higher scores to their attempts at queries in Kaleidoquery than in OQL ($t\ 5.98 : p = 0.000$).

4.3 Questionnaire

Each section of the questionnaire (overall reaction, query language constructs, query language comparison and subject comments) will be analysed and discussed.

4.3.1 Overall Reactions

Three questions were asked to ascertain the subjects overall reactions to a language after they had completed the five test queries on that language. Programmers showed no preference towards a language, whereas there was a significant preference by the non-programmers for Kaleidoquery ($t\ 2.28 : p = 0.049$). For each of the groups, the non-programmers disliked OQL to a significantly higher level than the programmers ($t\ 3.22 : p = 0.005$), but there was no significant difference in the liking of Kaleidoquery between the groups.

4.3.2 Query Language Constructs

Table 5 indicates the five language constructs that the subjects were asked to rate for ease of use for a specific language. The scores given were an independent rating of a language and its constructs, not a comparison of the two languages. The results show that programmers showed a preference for attribute comparison in OQL, while non-programmers showed a preference for selecting attributes and structuring results in Kaleidoquery.

4.3.3 Query Language Comparison

The subjects were asked to compare the two query languages, after they had completed both tests, and rate which if any they preferred, and to what extent they preferred it. Table 6 identifies which aspects of the languages subjects preferred to a significant level. It can be seen that non-programmers showed a significant preference for all of the Kaleidoquery language constructs that were tested except for data flows. It was found that programmers showed a significant preference of the data flow construct of the Kaleidoquery language.

4.3.4 Subject Comments

Comments from subjects essentially reflected their preference or dislike for a query language. The following are some of the comments in favour of Kaleidoquery:

Language factor	Programmers	Non-programmers
1. Selecting attributes		$K (t 2.61 : p = 0.028)$
2. Attribute comparison	$O (t 2.41 : p = 0.039)$	
3. Data flow		
4. Relationships		
5. Structuring		$K (t 3.24 : p = 0.010)$

Table 5: Language factors found significantly better when comparing the individual scores for the query language factors.

Language factor	Programmers	Non-programmers
1. Selecting attributes		$K (t 2.63 : p = 0.027)$
2. Attribute comparison		$K (t 3.12 : p = 0.012)$
3. Data flow	$K (t 2.67 : p = 0.026)$	
4. Relationships		$K (t 6.82 : p = 0.000)$
5. Structuring		$K (t 3.50 : p = 0.007)$
6. Building filters/constraints		$K (t 3.29 : p = 0.009)$
7. Overall comparison		$K (t 3.98 : p = 0.003)$

Table 6: Language factors found significantly better with subject comparison of Kaleidoquery and OQL via the questionnaire

Kaleidoquery is much easier than OQL. It is very clear and makes sense. It is easier to master and more interesting.

Kaleidoquery was much easier to use than OQL and the queries much clearer.

[OQL is] Very difficult for the computer phobic.

OQL is obviously a descendent from SQL, whose constructs reflect the way the database works, rather than how the user would word a query in natural language.

I think it [Kaleidoquery] simulates the way I'd think to query a database.

The final two are interesting to note, in that the first highlights a problem with textual query languages and the second notes that the Kaleidoquery language is a visual reflection of how users could visualise the query internally. There were few comments in favour of OQL.

I don't really like graphical languages—at least I've never found one that I think is easy or succinct as that makes it clear what's going on.

OQL was easier for me, possibly because I'm such a *verbal* person!

It is true that textual languages can be more succinct, but as we have seen in this evaluation, programmers were no better at creating textual language queries than they were at visual queries, even though they are very familiar with programming using textual languages. Also, programmers answered more *essentially correct* queries in Kaleidoquery than with OQL.

4.3.5 Discussion

The main points noted from the questionnaires can be summarised as:

- overall reactions—non-programmers preferred Kaleidoquery.
- query language constructs—regarding the independent evaluations of each of the language constructs there were only a few significant results, with programmers preferring attribute comparison in OQL, and non-programmers preferring the selection and structuring aspects of Kaleidoquery.
- query language comparison—the higher rating of attribute comparison in OQL by programmers did not cross over to their preferential rating of the two languages. Programmers did show a preference for data flows and non-programmers showed a preference for all the other aspects of Kaleidoquery.
- subject comments—these mostly reflected subjects preference for a specific language, and this tended to be Kaleidoquery.

5 Related Work

Visual query languages have attempted to bridge the gap of usability for users. Forms based query languages such as QBE [46], present the database structure as tables or forms into which queries can be placed. Graph based query languages, such as Guidance [18], have the advantage over forms style interfaces that they can directly represent relationships within the structure of the database and the query. Icon based languages (e.g. Iconic Browser [41]) represent database concepts pictorially and can allow for the direct manipulation of icons to represent queries. Multi paradigm query interfaces also exist to allow the user to pick and choose or alternate between interface styles [7, 13]. Visual query languages are now also incorporating three dimensional aspects into the query language (e.g. AMAZE [4]), but with AMAZE constraints are still specified via a forms interface. In addition, AMAZE is much less powerful than Kaleidoquery (see Table 7). Many advances are being made in the area of dynamic query visualisation (e.g. Spotfire [1]), though these tend to be limited in the expressive power of their queries and the range and type of information they can query. There are difficulties in finding a user friendly approach for defining a query language with high expressive power without resorting to visualising the query as a mapping from the textual way of expressing the query, see [24, 27].

The filter flow model for depicting the boolean *and* and *or* relations had a prototype constructed and evaluated [45]. The prototype was not connected to a real database but used a single relation over which users could select the attribute they wished to constrain. From this operation they obtained a list of all the instances of that attribute in the table. Instances that the user was interested in could then be selected. Furthermore, the thickness of the data flow was altered to reflect the size of the results, although with the prototype actual values were not used but rather the thickness was determined heuristically. The early evaluations of this form of depiction of the boolean operators has shown promise, and so was adopted for the Kaleidoquery language and extended so as to be used for a real database. Current query languages for ODMG compliant object databases are limited to OQL interfaces, and the visual interfaces of QUIVER [11] and GOQL [23].

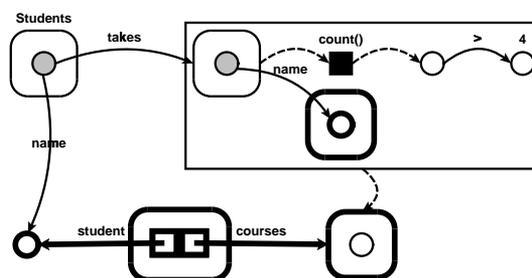


Figure 20: Example QUIVER query, from [11]

Only two simple examples of the QUIVER query language are given in [11], but preliminary evaluations show that it is easier to use than the standard textual OQL interface. Figure 20 shows an example QUIVER query that finds students who take more than four courses, in each case returning the name of the student along with the collection of names of the courses the student takes. In the top left of Figure 20 we begin with the extent of Students and follow the relationship *takes* that yields a set of courses to which the operation *count* is applied and this value is then tested to see if it is greater than 4. Where the size of the collection is greater than 4, the names of the courses are output and this is combined with the name of the student taking the course, shown at the bottom of the query.

GOQL represents the class and its attributes as a list or table and uses links and diagrammatic notation to form a query, as illustrated in Figure 21. A complete description of its constructs is given in [23]. Figure 21 shows an example of a GOQL query that finds all female nurses who are either students or supervisors. Neither of the papers describe the interface that has been implemented to support the visual languages.

Other visual query languages to object databases include PESTO [6], OdeView [12], and ADAM [14]. These are implemented on top of object databases that use their own object query language, and as such are not ODMG compliant. PESTO and OdeView take the form of query browser interfaces. These allow the user to select a starting collection and browse the instances and their relations. Queries are forms based in that the users use the template of the instance view to create a query. ADAM is a multi-paradigm query interface allowing queries to be created in text, forms or graph styles.

A classification for database query interfaces is described in [28]. In this case we are only comparing the query

Query Creation and Visualisation

Interface	Visualisation Style	Expressive Power	Evaluation	and/or premisses complexity	universal quantification	GROUP BY	structuring results	relationship queries
O ₂ [3]	textual	see ODMG OQL allows for all 12 main features plus collection operations such as list accessors and conversion and constructor operations						
Kaleidoquery [31]	data flow graph	all 12 main features defined in Kaleidoquery.	see work reported here	✓			✓	
QUIVER [11]	diagrammatic graph query representation	selection, simple mapping on integer and string values, projection, aggregates, path expressions, joins	a limited user evaluation was conducted [11] that concluded that it was easier to formulate correct queries with the QUIVER interface than the OQL interface					✓
GOQL [23]	tabular class representation with textual & diagrammatic query attachments	all features except for the <i>structuring</i> and <i>grouping</i> of results are defined in the query language description						✓
PESTO [6]	form display of the class attributes allowing mappings to be placed on attributes and queries on related classes showing as links joining the classes	all main features except for <i>structuring results</i> and <i>ordering</i> and <i>grouping results</i> are supported						✓
OleView [12]	table style display of selection predicates defining the operands and the operation associated with them; for projection a list of attributes is displayed for the user to highlight their selection	all features except for <i>membership testing</i> and <i>universal and existential quantification</i> , <i>set operations</i> , <i>joins</i> , <i>arithmetic operators</i> , <i>structuring results</i> and <i>ordering</i> and <i>grouping results</i> are defined						?
ADAM multi-paradigm query interface [14]	textual, forms based and graph based representations	<i>selection</i> , <i>simple mappings</i> , <i>projection</i> , <i>boolean queries</i> , <i>aggregates</i> and <i>path expressions</i> are supported	the three query interfaces of ADAM were evaluated with users expressing an overwhelming preference for the graph based interface; this preference is backed up their performance scores; the need for a query history was identified in the evaluation [13]					✓

Table 7: Query Creation and Visualisation Classification

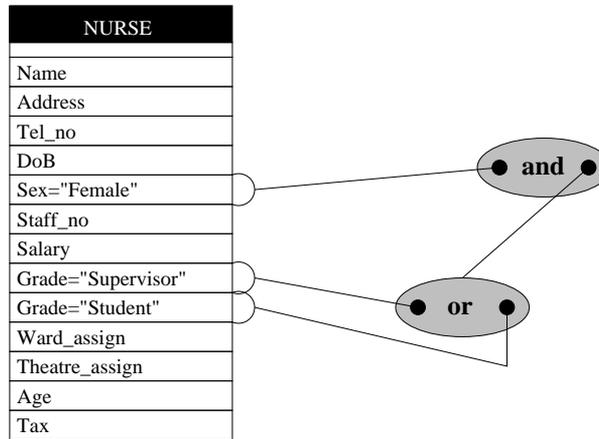


Figure 21: Example GOQL query, from [23]

language aspects of a query interface. The languages will be described and classified under the following areas:

- Visualisation style—this describes how the query is visualised. The different forms of visualisation that a query could use are textual, form, graph/diagrammatic, iconic and multi-paradigm.
- Expressive Power—this defines the types of queries that can be created with the interface. Catarci *et al* [8] used and extended a previous classification by Chandra [10] known as Chandra’s hierarchy. This hierarchy can be seen in Figure 22 with the extended classes of Catarci *et al* [8] shown as gray hatched boxes. It was noted by Catarci *et al* [8] that most visual query systems are less expressive than relation algebra, and very few are placed in the upper levels. It was also noted in [8] that it can be very hard to classify the visual query system as formal, or even precise specifications and proofs for visual query systems are rare. Due to this lack of formal descriptions for visual query systems, and the current tendency for visual query languages to have limited expressive power, we identify key areas that query languages address. The main features that a visual query system exhibits include (but are not limited to):
 1. *Selection*: selecting an entity from the database.
 2. *Simple mapping*: finding data values that are associated with another known data value.
 3. *Projection*: the data values that are required from an entity.
 4. *Boolean queries*: conjunctions and disjunctions used within mappings.
 5. *Set operations*: use of union, intersection and set difference between mappings.
 6. *Aggregates*: applying aggregates such as sum, count, etc. to data values, etc.
 7. *Path expressions*: navigating between attributes.
 8. *Membership Testing and Universal and Existential Quantification*.
 9. *Joins*.
 10. *Structuring results*: structuring the output of the results of separate entities.
 11. *Ordering and Grouping results*.
 12. *Arithmetic operators*: allowing arithmetic to be performed on numeric attributes.
- Formal specification—this identifies whether the syntax and semantics of the visual query language has been formally specified.
- Evaluation—this lists any details on the evaluation of the query interface.
- Handling of identified language problems—in this section is listed problems identified with previous query languages. With this list we can check whether current query interfaces address these identified problems. The query language problems were defined in Section 2.1.

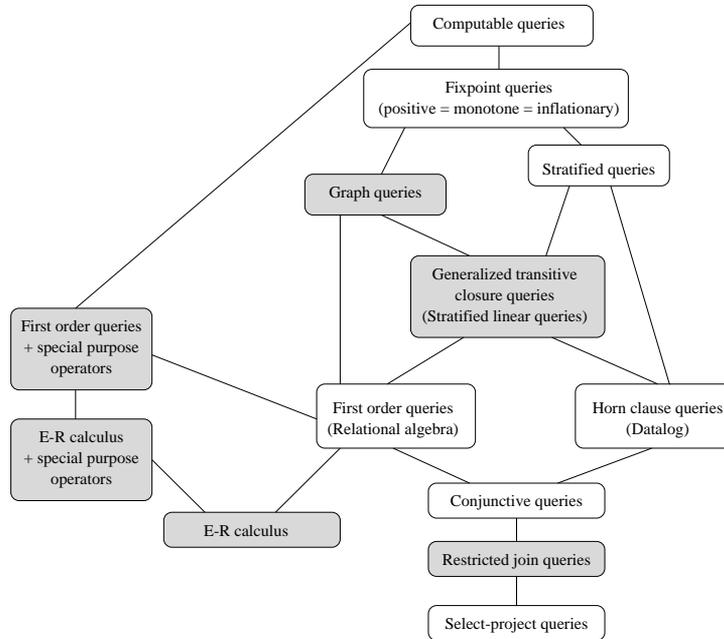


Figure 22: Classes of queries in the enriched Chandra’s hierarchy, from [8]

Using this classification we can compare the Kaleidoquery language with other visual query languages for object databases. In this context we compare the Kaleidoquery language on those factors that are independent of an implementation. Certain aspects of the key language problems from Section 2.1 have been omitted as they relate to the implementation of the language, as in this case we are only comparing the query languages, and not their interfaces.

- Visualisation style—the visualisation styles reviewed run the gamut of styles from textual through forms and graph based to multi-paradigm query languages and query browsers. Kaleidoquery is the only language that allows iconic representations of the classes; all other languages represent the classes textually.
- Expressive Power—as stated previously, visual query languages offer a range of basic query language operations (see Table 7). It is also true to say that it is common for all papers to describe query language capabilities inadequately and it is not always clear what the full range of supported operations are. In comparing the languages expressive powers, we can see that Kaleidoquery is the only visual language to offer the 12 main query features defined previously. Of the three interfaces to ODMG OQL, GOQL supports all main features except for structuring and grouping of results, whereas QUIVER offers a very limited set of query facilities.
- Formal specification—none of the query languages has a formal specification associated with it.
- Evaluation—QUIVER and ADAM are the only query languages in this review that have had evaluations of their interfaces reported. QUIVER was evaluated but this was in combination with its interface—there was no comparison done between QUIVER and OQL independent of an interface. The only conclusions drawn from the evaluation was that it was easier to formulate correct queries in QUIVER than in OQL. This was done by scoring 8 queries attempted by 7 participants for QUIVER and 6 for OQL. With ADAM, the evaluations conducted were to assess whether users showed any preference in the query interface they used and whether there were any differences in query times, scores, etc. A problem with these evaluations is that they only conclude whether one interface is better, in some way, e.g., easier to formulate correct queries, than another interface. The evaluations do not have hypothesis that attempt to examine why this might be the case.
- Key language problem areas—Regarding the language problems identified in Section 2.1, none of the papers directly mention these problems. The query languages do offer visual support for queries over relationships, but this is the only problem area in which it can be seen that they are offering an alternative to the textual

representation. Furthermore, the languages do not evaluate this aspect of their query language. Kaleidoquery attempts to address four out of the seven problem areas. Kaleidoquery has not fully addressed the areas of universal quantification, `group by`, and `sum` and `count`.

- universal quantification—Kaleidoquery has visualised the `for all` clause of OQL by tagging the data flow with the term *all* (see Section 2.11) to attempt to clarify the nature of the expression.
- `group by`—the Kaleidoquery language uses the term `group` rather than `group by` and visualises the results as moving from a set to a set of sets, with the partitioning conditions shown as separate data flows.
- `sum` and `count`—these operations have not had their names altered but, as suggested in Section 2.1, system constraints and feedback could be used to reduce confusion of these operations.

Of the work reviewed, Kaleidoquery offers the most expressive power, and attempts to address more of the key problem areas in query languages. Its evaluation is also more comprehensive and has built in hypotheses that attempt to evaluate specific aspects of the query language.

6 Conclusion

6.1 Kaleidoquery

We have described the design of Kaleidoquery, a graphical query language for the querying of object databases. We have described the features of OQL and shown how they are supported by Kaleidoquery. Queries produced by Kaleidoquery can be translated into OQL and then submitted to an object database, so a Kaleidoquery interface implementation should be compatible with any ODMG compliant database. The design philosophy behind Kaleidoquery has been to design a language that overcomes previously identified problems in textual and visual query languages, and also to identify and overcome areas of concern in OQL. Each of the tasks that a user may wish to perform have been separated in our visualisation.

- The results of a query are shown at the top of the query. With OQL the `select` statements can be nested when the user wants to structure their results. Our method makes it easier to see the results of the query at a glance and also to restructure the results, without the need to completely rewrite the query, as in OQL.
- The ordering of results is also visualised at the top of a query, so that users can see at a glance whether an ordering has been applied to the results, rather than requiring to examine the OQL for any nested ordering clauses.
- Relationships are visualised in the same manner regardless of their cardinalities, unlike in OQL.

6.2 Evaluation

The results from each of the subject groups will be discussed in turn.

6.2.1 Programmers

There were very few statistically significant results for the programmers.

1. As regards query language constructs, programmers correctly defined significantly more relationship queries in Kaleidoquery than OQL.
2. Programmers showed a higher preference for attribute comparison in OQL in the independent language evaluation, but this preference did not appear when they were asked to rate the query languages together.
3. Programmers showed a significant preference for data flows over *and* and *or* in OQL.

6.2.2 Non-programmers

The non-programmers produced many significant results:

1. Non-programmers answered significantly more essentially correct queries in Kaleidoquery than OQL.
2. They defined significantly more correct queries with results selection, attribute comparison and relationships language constructs.
3. In their self assessment scores, they scored their queries to a significantly higher level in Kaleidoquery than in OQL.
4. They showed a higher preference for the selection and structuring aspects of Kaleidoquery. These preferences did cross over to the query language comparison, where non-programmers preferred all the language aspects except for data flows.

6.2.3 Discussion

As could be expected, programmers performed better than non-programmers in each of the languages, although this difference was lessened with Kaleidoquery. The differences between the performance of each of the groups was significant. A further evaluation could be conducted on the programmer group to assess their abilities with more complex queries.

The non-programmers were unfamiliar with textual languages, and found it difficult to achieve the desired competency in OQL, but did manage to correctly answer almost half of the queries posed with Kaleidoquery. Several methods can be envisaged to solve this problem:

1. design an unambiguous natural language interface;
2. provide extensive syntax correction interfaces for textual query languages;
3. design a query language that visualises the query in a clearer more understandable manner.

With Kaleidoquery it is the final path that has been taken to satisfying this problem. With the comparison of Kaleidoquery and OQL we have shown that non-programmers prefer the language, can write and attempt to write more correct queries, consider themselves to have done better, and prefer nearly all the language constructs of Kaleidoquery when compared to OQL. Of the visual query interfaces to ODMG compliant object databases, Kaleidoquery offers the most expressive power, and attempts to address more of the key problem areas in query languages.

There are many advances being made in query languages, such as support for semi-structured data, deduction, spatial reasoning, etc. Unfortunately the same cannot be said for the evaluation of the languages and their interfaces. Where query languages and interfaces have been evaluated, evaluations that have little rigour have often been conducted, with simple hypothesis comparing whether users can obtain more correct queries with one language compared to another. Even then, the evaluations compare query languages and interfaces at the same time, confusing the evaluation and therefore giving inconclusive results, as it cannot be decided whether one query language is better than another as the interface is influencing the results.

Experiments have to be structured to test hypotheses to see whether certain aspects of the language prove to be problematic and whether a redesign of these aspects of the language solves the problems. This can only be done through evaluation, such as comparing *and* and *or* with the filter flow method as done by Young and Shneiderman[45], or the use of sub-tests within a larger evaluation as performed in the evaluation of Kaleidoquery. A systematic approach to the design of an evaluation will probably uncover more problems, in a shorter time, than an unstructured approach.

In future, evaluations need to be conducted that have more rigour, with hypotheses that will aid the redesign of the language such that they: identify language problems, such as unusually difficult functions or common errors; identify what is causing the problems; and suggest improvements and redesigns of the language to attempt to solve these problems. In general the area of database interfaces, and more specifically the area of query language interfaces to databases, remains a largely unresearched topic; only when a user centred approach with rigorous evaluations takes place can the full potential of the advances in databases be usefully realised by the users.

References

- [1] Christopher Ahlberg. Spotfire: An Information Exploration Environment. *SIGMOD Record*, 24(4):25–29, December 1996.
- [2] Christine L. Borgman. The User’s Mental Model of an Information Retrieval System; An Experiment on a Prototype Online Catalog. *International Journal of Man-Machine Studies*, 24:47–64, 1986.
- [3] P. Borras and et al. Building User Interfaces for Database Applications: The O2 Experience. In *SIGMOD RECORD*, pages 32–38, 1992.
- [4] J. Boyle, S. Leishman, and P. M. D. Gray. From WIMP to 3D: the development of AMAZE. *Journal of Visual Languages and Computing*, 7:291–319, 1996.
- [5] M. D. S. Braine. On the Relation Between the Natural Logic of Reasoning and Standard Logic. *Psychological Review*, 85:1–21, 1978.
- [6] Michael Carey, Laura Haas, Vivek Maganty, and John Williams. PESTO: An Integrated Query/Browser for Object Databases. In *Proc. of the 22nd VLDB*, pages 203–214, Mumbai(Bombay), India, 1996.
- [7] T. Catarci, S. K Chang, and G. Santucci. Query Representation and Management in a Multiparadigmatic Visual Query Environment. *Journal of Intelligent Information Systems*, 3(3):299–330, 1994.
- [8] Tiziana Catarci, Maria F. Costabile, Stefano Levialdi, and Carlo Batini. Visual Query Systems for Databases: A Survey. *Journal of Visual Languages and Computing*, 8:215–260, 1997.
- [9] R. G. G. Cattell, Douglas Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland, and Drew Wade. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Inc., 1997.
- [10] A. K. Chandra. Theory of Database Queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–9, 1988.
- [11] Manoj Chavda and Peter Wood. Towards an ODMG-Compliant Visual Object Query Language. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *Proceedings of 23rd International Conference on Very Large Data Bases*, pages 456–465, 1997.
- [12] S. Dar, N. H. Gehani, H. V. Jagadish, and J. Srinivasan. Queries in an Object-Oriented Graphical Interface. *Journal of Visual Languages and Computing*, 6(1):27–52, 1995.
- [13] D. K. Doan, N. W. Paton, and A. C. Kilgour. Design and User Testing of a Multi-paradigm Interface to an Object-Oriented Database. *ACM SIGMOD Record*, 24(3):12–17, September 1995.
- [14] D. K. Doan, N. W. Paton, A. C. Kilgour, and G. Al-Qaimari. A Multi-Paradigm Query Interface To An Object-Oriented Database. *Interacting With Computers*, 7(1):25–47, 1995.
- [15] S. Fillenbaum. Mind Your P’s and Q’s: The Role of Content and Context in some uses of And, Or, and If. In Gordon Bower, editor, *The Psychology of Learning and Education*, volume 11. Academic Press, New York, 1977.
- [16] David Greenblatt and Jerry Waxman. A Study of Three Database Languages. In *Proceedings of the International Conference on Databases: Improving Usability and Responsiveness*. Academic Press, 1978.
- [17] S. Greene, S. Devlin, P. Cannata, and L. Gomez. No IFs, ANDs, or ORs: A Study of Database Querying. *International Journal of Man-Machine Studies*, 32:303–326, 1990.
- [18] David Haw, Carole Goble, and Alan Rector. GUIDANCE: Making it Easy for the User to be an Expert. In *Proc. 2nd Int. Workshop On Interfaces to Database Systems*, pages 19–43. Springer-Verlag, 1994. P. Sawyer (Ed).
- [19] Matthias Jarke, Jon A. Turner, Edward A. Stohr, Yannis Vassiliou, Norman H. White, and Ken Michielsen. A Field Evaluation of Natural Language for Data Retrieval. *IEEE Transactions on Software Engineering*, 11(1):97–114, 1985.

- [20] P. Johnson. *Human Computer Interaction*. McGraw-Hill, London, 1992.
- [21] M. A. Just. Comprehending Quantified Sentences: The Relation Between Sentence-Picture and Semantic Memory Verification. *Cognitive Psychology*, 6:216–236, 1974.
- [22] C. Katzeff. Dealing with a Database Query Language in a New Situation. *International Journal of Man-Machine Studies*, 25:1–17, 1986.
- [23] Euclid Keramopoulos, Philippos Pouyioutas, and Chris Sadler. GOQL, a Graphical Query Language for Object-Oriented Database Systems. In *Basque International Workshop on Information Technology*, pages 35–45, 1997.
- [24] M. Kuntz and R. Melchert. Pasta-3's Graphical Query Language: Direct Manipulation, Cooperative Queries, Full Expressive Power. In *Proceedings of the 15th International Conference on Very Large Databases*, 1989.
- [25] L. Linde and Y. Waern. On Search in an Incomplete Database. *International Journal of Man-Machine Studies*, 22:563–579, 1985.
- [26] A. Michard. Graphical Presentation of Boolean Expressions in a Database Query Language: Design Notes and an Ergonomic Evaluation. *Behaviour and Information Technology*, 1(3):279–288, 1982.
- [27] T. Miura and K. Moriya. On the Completeness of Visual Operations for a Semantic Data Model. *Data and Knowledge Engineering*, 9:19–44, 1992.
- [28] Norman Murray. *A Visual Query Language and its Implementation in a 3D Environment*. PhD thesis, Department of Computer Science, The University of Manchester, 1999.
- [29] Norman Murray, Carole Goble, and Norman Paton. A Framework for Describing Visual Interfaces to Databases. *Journal of Visual Languages and Computing*, 9(4):429–256, August 1998.
- [30] Norman Murray, Carole Goble, and Norman Paton. Kaleidoscope: A 3D Environment for Querying ODMG Compliant Databases. In Yannis Ioannidis and Wolfgang Klas, editors, *Proceedings Visual Databases 4*, pages 85–101. Chapman & Hall, London, 1998.
- [31] Norman Murray, Norman Paton, and Carole Goble. Kaleidoquery: A Visual Query Language for Object Databases. In Tiziana Catarci, Maria F. Constabile, Giuseppe Santucci, and Laura Tarantino, editors, *Proceedings Advanced Visual Interfaces*, pages 247–257. ACM Press, 1998.
- [32] William M. Newman and Michael G. Lamming. *Interactive System Design*. Addison-Wesley, 1995.
- [33] Phyllis Reisner. Use of Psychological Experimentation as an Aid to Development of a Query Language. *IEEE Transactions on Software Engineering*, 3(3):218–229, May 1977.
- [34] Phyllis Reisner. Formal Grammar and Human Factors Design of an Interactive Graphics System. *IEEE Transactions on Software Engineering*, 5(2):229–240, 1981.
- [35] Phyllis Reisner. Query Languages. In M. Helander, editor, *Handbook of Human-Computer Interaction*, chapter 12, pages 257–280. Elsevier Science Publishers B. V., North-Holland, 1988.
- [36] Phyllis Reisner, Raymond F. Boyce, and Donald D. Chamberlain. Human Factors Evaluation of Two Database Query Languages—Square and Sequel. In *AFIPS Proceedings*, pages 447–452. AFIPS Press, NJ, 1975.
- [37] Ben Shneiderman. Improving the Human Factors Aspect of Database Interactions. *ACM Transactions on Database Systems*, 3(4):417–439, December 1978.
- [38] Ben Shneiderman. Visual user interfaces for information exploration. In *Proceedings of the 54th Annual Meeting of the American Society for Information Science*, pages 379–384, Medford, NJ, 1991. Learned Information Inc.
- [39] D. W. Small and L. J. Weldon. An Experimental Comparison of Natural and Structured Query Languages. *Human Factors*, 25:253–263, 1983.

- [40] John C. Thomas and John D. Gould. A Psychological Study of Query By Example. In *AFIPS Proceedings*, pages 439–445. AFIPS Press, NJ, 1975.
- [41] K. Tsuda, M. Hirakawa, M. Tanaka, and T. Ichikawa. Iconic Browser: An Iconic Retrieval System for Object-Oriented Databases. *Journal of Visual Languages and Computing*, 1(1):59–76, 1990.
- [42] P. C. Wason and P. N. Johnson-Laird. *Psychology of Reasoning*. Harvard University Press, Cambridge, MA, 1972.
- [43] C. Welty. Correcting User Errors in SQL. *International Journal of Man-Machine Studies*, 22:463–477, 1985.
- [44] Charles Welty and David W. Stemple. Human Factors Comparison of a Procedural and a Nonprocedural Query Language. *ACM Transactions on Database Systems*, 6(4):626–649, December 1981.
- [45] Degi Young and Ben Shneiderman. A Graphical Filter/Flow Representation of Boolean Queries: A Prototype Implementation and Evaluation. *Journal of the American Society for Information Science*, 44(6):327–339, 1993.
- [46] M. Zloof. Query-By-Example: A Data Base Language. *IBM Systems Journal*, Vol. 4, pages 324–343, December 1977.

A Evaluation Queries and Answers

A.1 OQL

Query 1 Find the name and salary of people who are aged less than 50.

```
select tuple(name:p.name, salary:p.salary)
from p in People
where p.age < 50
```

Query 2 Find the names of people who work for “IBM” or “DEC” along with those that have a salary of more than 35000.

```
select p.name
from p in People
where (p.employer.name = ``IBM``
      or p.employer.name = ``DEC``)
      and p.salary > 35000
```

Query 3 Find orders where the supplier is called “IBM” and the total cost of the order is larger than 2000.

```
select o
from o in Orders
where o.supplier.name = ``IBM``
      and o.totalcost > 2000
```

Query 4 Find pairs of invoice numbers where each invoice has the same total cost.

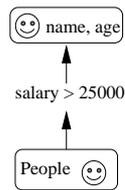
```
select tuple(
      invoicenumero:o.invoicenumero,
      invoicenumberP:p.invoicenumber)
from o in Orders, p in Orders
where o.totalcost = p.totalcost
```

Query 5 Obtain the locations of companies called “IBM” with a list of the employees that earn more than 30000.

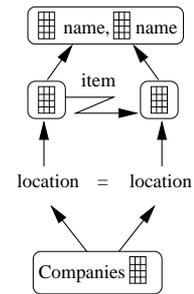
```
select tuple(location:c.location,
      theEmployees:(select e
      from e in c.employees
      where e.salary > 30000))
from c in Companies
where c.name = "IBM"
```

A.2 Kaleidoquery

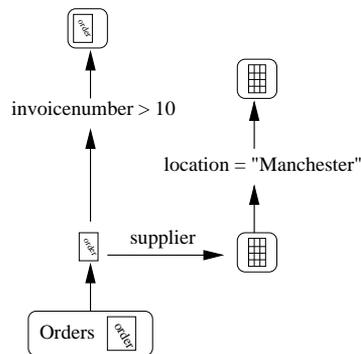
Query 1 Find the name and age of all people earning more than 25000.



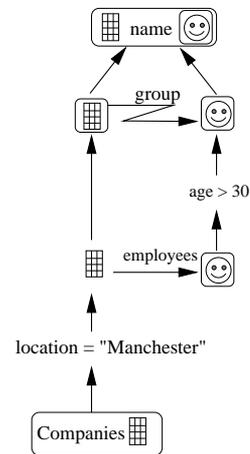
Query 4 Find pairs of names of companies that have the same location.



Query 2 Find the orders where the supplier is located in Manchester and the invoice number is greater than 10.



Query 5 Find the names of companies located in Manchester with a list of their employees who are aged over 30.



Query 3 Find the names of people who work for companies located in Manchester or London and who are aged more than other people.

