A Tutorial on the IQL Query Language

AutoMed Technical Report No. 28, Version 1.2

Alex Poulovassilis and Lucas Zamboulis, July 2008

1 Introduction

IQL is a typed, functional language. It supports strings (e.g. 'Computer Science', 'Fred''s car'), booleans (True, False), real numbers (e.g. 10.51) and integers. It also supports tuples (e.g. {1,2,3}), enumerated lists (e.g. `[5, 5, 6, 7, 7, 7]`), bags (e.g. `B[5, 5, 6, 7, 7, 7]`) and sets (e.g. `S[5, 6, 7]`) of homogeneous values (i.e. values of the same type). Tuples, lists, sets and bags can be arbitrarily nested, e.g. `[{[1,2,3],[4,5]}, {[3,4],[1,2,5]}, {[],[6,7,8]}]` is a list of pairs, each of whose components is a list of numbers.

A number of built-in functions and operators are provided e.g. `(+)`, `(-)`, `(*)`, `(/)`, `(=)`, `(!=)`, (<), (>), (<=), (>=), `and`, `or`, `not` — see Appendix B for a list of these and a description of each one.

The binary built-in operators are supported both in prefix form, where they need to be enclosed in round brackets as shown above, and in infix form, where they are used without round brackets. For example, the expressions `(+) 5 6` and `5 + 6` are equivalent.

New, anonymous, functions can be defined by using **lambda abstractions**. For example:

```
lambda  {x,y,z} ((x+y)*z)
```

defines a function which takes a triple, adds the first two components and multiplies by the third one. So

```
(lambda  {x,y,z} ((x+y)*z)) {3,4,5}
```

returns `(3+4)*5` = 35

Also supported are 'let' expressions, of the form `let v = e1 in e2`, for any expressions `e1` and `e2`. These translate internally into an expression of the form

```
(lambda v e2) e1
```

For example, the query

```
let v = (200 + 500) in (v * v)
```

returns `700 * 700` = 490000, the query

```
let f = (lambda  {x,y,z} ((x+y)*z)) in ((f {1,2,3}) + (f {3,4,5}))
```

returns `((1+2)*3) + ((3+4)*5)` = 44.

All IQL queries are expressed internally in the form of Abstract Syntax Graphs (ASGs) — for details, see the AutoMed technical report by Jasper, Poulovassilis, Zamboulis et al. on "Processing IQL Queries in the AutoMed Toolkit".

As discussed earlier, IQL supports the **list**, **bag** and **set** collection types. To remove duplicates from lists and bags, a function `distinct` is provided. We give some examples of IQL queries below, which illustrate the use of most of its features.

2 Expressing the Relational Algebra in IQL

Consider two collections R and S both consisting of 3-tuples of numeric values. We can compute the **union** of R and S by:

```
R ++ S
```

If R and S are lists, then `++` simply appends S to the end of R.

The operator `++` does not eliminate duplicates from its result if R and S are bags or lists. To do so, we can use the `distinct` function:

```
distinct (R ++ S)
```

or, equivalently, we can use the `union` function:

```
R union S
```

We can compute the **difference** of R and S by:

```
R -- S
```

For each member r of R, this operator returns $max(0,occurs(\mathrm{r},\mathrm{R}) - occurs(\mathrm{r},\mathrm{S}))$ instances of r in the result (where $occurs(\mathrm{r},\mathrm{R})$ denotes the number of instances of r in R). If R and S are lists, then the result is produced in ascending sorted order. If R and S are sets, the result is just ordinary set difference.

To return members of R which do not appear in S, we can use

```
[r | r <- R; not (member S r)]
```

if R and S are lists. This is an example of a *list comprehension*.

If R and S are bags, the same expression is used but starting with "`B[` " instead of "`[` ". If R and S are sets, then "`S[` " is used instead of "`[` "[1].

We can compute the **intersection** of R and S by:

```
R intersect S
```

For any value v, this returns $min(occurs(\text{v,R}),occurs(\text{v,S}))$ instances of v in the result. If R and S are lists, then the result is produced in ascending sorted order. If R and S are sets, the result is just ordinary set intersection.

Henceforth in this tutorial, we will assume that R and S are lists, but our examples generalise, as illustrated above, to the cases of bag and set collections.

To return members of R which also appear in S, we use:

```
[r | r <- R; member S r]
```

[1]The general syntax of list comprehensions is

$$[e \mid Q_1;\ \ldots;\ Q_n]$$

Here, e is an expression, termed the *head* of the comprehension, and Q_1 to Q_n are *qualifiers*, where $n \geq 0$. Each qualifier is either a *filter* or a *generator*. Generators have syntax $p \leftarrow s$, where p is a pattern and s is a list-valued expression. A *pattern* is an expression consisting of constructors and variables only (no functions or lambda abstractions). The variables of p are successively bound by iterating through s. Any variables appearing in the head e inherit these bindings. A filter is a boolean-valued expression, which must be satisfied by the values generated by the generators in order for these values to contribute to the final result of the comprehension. Comprehensions are a convenient syntax and add no extra expressiveness to languages such as IQL since they translate into applications of the $flatmap$ and if functions:

$$\begin{array}{lcl} [e \mid p \leftarrow s;\, Q] & \equiv & flatmap\ (lambda\ p\ [e \mid Q])\ s \\ [e \mid e';\, Q] & \equiv & if\ e'\ [e \mid Q]\ [\,] \\ [e \mid\,] & \equiv & [e] \end{array}$$

Bag comprehensions similarly have the syntax

$$B[e \mid Q_1;\ \ldots;\ Q_n]$$

and in this case any generators within the Q_i are of the form $p \leftarrow s$ where s is a bag. They also translate into applications of $flatmap$ and if internally:

$$\begin{array}{lcl} B[e \mid p \leftarrow s;\, Q] & \equiv & flatmap\ (lambda\ p\ B[e \mid Q])\ s \\ B[e \mid e';\, Q] & \equiv & if\ e'\ B[e \mid Q]\ BNil \\ B[e \mid\,] & \equiv & (BConseBNil) \end{array}$$

Similarly, set comprehensions have the syntax

$$S[e \mid Q_1;\ \ldots;\ Q_n]$$

and any generators within the Q_i are of the form $p \leftarrow s$ where s is a set. They also translate into applications of $flatmap$ and if, as follows:

$$\begin{array}{lcl} S[e \mid p \leftarrow s;\, Q] & \equiv & flatmap\ (lambda\ p\ S[e \mid Q])\ s \\ S[e \mid e';\, Q] & \equiv & if\ e'\ S[e \mid Q]\ SNil \\ S[e \mid\,] & \equiv & (SConseSNil) \end{array}$$

We can also use comprehension syntax to express **projection**. For example, we can project the first and third components of tuples in R by:

```
[{x,z} | {x,y,z} <- R]
```

This query can be read as stating:

```
for every 3-tuple {x,y,z} in R do
    return {x,z}
```

We can also use comprehension syntax to express **selection**. For example, to return tuples of R whose second component is greater than 10:

```
[{x,y,z} | {x,y,z} <- R; y > 10]
```

This query can be read as stating:

```
for every 3-tuple {x,y,z} in R do
    if y > 10
    then return {x,z}
```

We can also use comprehension syntax to express **cartesian product**. For example to return the cartesian product of R, S and a third relation T consisting of 1-tuples:

```
[{x1,y1,z1,x2,y2,z2,x3} | {x1,y1,z1} <- R; {x2,y2,z2} <- S; {x3} <- T]
```

This query can be read as stating:

```
for every 3-tuple {x1,y1,z1} in R do
    for every 3-tuple {x2,y2,z2} in S do
        for every 1-tuple {x3} in T do
            return {x1,y1,z1,x2,y2,z2,x3}
```

Finally, we can use comprehension syntax to express **joins**. For example, to join R and S over their second column and return a set of 6-tuples:

```
[{x1,y1,z1,x2,y2,z2} | {x1,y1,z1} <- R; {x2,y2,z2} <- S; y1 = y2]
```

This query can be read as stating:

```
for every 3-tuple {x1,y1,z1} in R do
    for every 3-tuple {x2,y2,z2} in S do
        if y1 = y2
        then return {x1,y1,z1,x2,y2,z2}
```

To eliminate the duplicated join column from the above result and return a set of 5-tuples:

```
[{x1,y1,z1,x2,z2} | {x1,y1,z1} <- R; {x2,y2,z2} <- S; y1 = y2]
```

To join R and S over their second and third columns, eliminating the duplicated join columns from the result:

```
[{x1,y1,z1,x2} | {x1,y1,z1} <- R; {x2,y2,z2} <- S; y1 = y2; z1 = z2]
```

To perform a theta-join of R and S such that the first component of tuples of R is less than the third component of tuples of S:

```
[{x1,y1,z1,x2,y2,z2} | {x1,y1,z1} <- R; {x2,y2,z2} <- S; x1 < z2]
```

To perform a more complex theta-join of R and S, illustrating the use of the `and`, `or` and `not` functions:

```
[{x1,y1,z1,x2,y2,z2} | {x1,y1,z1} <- R; {x2,y2,z2} <- S;
                       not (((x1 < z2) or (x1 > z2)) and (y1 = y2))]
```

2.1 Support for variable unification

IQL allows the same variable to appear within more than one pattern within a comprehension. It automatically renames such duplicate variable occurrences into unique variable names and adds the necessary equality constraints between the new variables. This makes writing IQL queries simpler and quicker. For example, the following list comprehension queries given earlier:

```
[{x1,y1,z1,x2,y2,z2} | {x1,y1,z1} <- R; {x2,y2,z2} <- S; y1 = y2]
[{x1,y1,z1,x2,z2} | {x1,y1,z1} <- R; {x2,y2,z2} <- S; y1 = y2]
[{x1,y1,z1,x2} | {x1,y1,z1} <- R; {x2,y2,z2} <- S; y1 = y2; z1 = z2]
```

can be written, equivalently, as follows by the user, and IQL will translate them automatically into the above longer forms for evaluation:

```
[{x1,y,z1,x2,y,z2} | {x1,y,z1} <- R; {x2,y,z2} <- S]
[{x1,y,z1,x2,z2} | {x1,y,z1} <- R; {x2,y,z2} <- S]
[{x1,y,z,x2} | {x1,y,z} <- R; {x2,y,z} <- S]
```

2.2 Support for nested queries

IQL queries can be arbitrarily **nested**, so the collections resulting from any of the above operations can form the input of another operation.

For example, suppose that R and S both have scheme (Name,Gender,Age). The IQL query corresponding to the following relational algebra expression:

$$\sigma_{Age>25}(\pi_{Name,Age}((\sigma_{Gender='Male'}R) \; - \; (\sigma_{Gender='Male'}S)))$$

is:

```
[{n,a} | {n,a}<-[{n,a} | {n,g,a} <- [{n,g,a}|{n,g,a}<-R; g = 'Male'] --
                                    [{n,g,a}|{n,g,a}<-S; g = 'Male']
                 ];
         a > 25]
```

We can simplify this by merging the outermost selection and projection operations:

```
[{n,a} | {n,g,a} <- [{n,g,a}|{n,g,a}<-R; g = 'Male'] --
                    [{n,g,a}|{n,g,a}<-S; g = 'Male'];
         a > 25]
```

This optimisation is known as **loop fusion**.

Exercise

Given that R and S both have scheme (Name,Gender,Age), construct an IQL query corresponding to the following relational algebra expression:

$$\sigma_{Name='Jones'\ AND\ Gender='Female'}(\pi_{Name,Gender}((\sigma_{Age>50}R)\ \cap\ (\sigma_{Age<50}S)))$$

Hint: start from the innermost sub-queries and construct the IQL query outwards.

3 Grouping and Aggregation Operations in IQL

IQL supports the expected grouping and aggregation operations. For example, to count the number of elements in R:

```
count R
```

to sort R:

```
sort R
```

to remove duplicates from R:

```
distinct R
```

To return the maximum value in R's second column:

```
max [y | {x,y,z} <- R]
```

and the minimum:

```
min [y | {x,y,z} <- R]
```

To return the sum over R's third column:

```
sum [z | {x,y,z} <- R]
```

and the average:

```
avg [z | {x,y,z} <- R]
```

To group R on its first column:

```
group [{x,{y,z}} | {x,y,z} <- R]
```

on its second column:

```
group [{y,{x,z}} | {x,y,z} <- R]
```

and on its second and third columns:

```
group [{{y,z},x} | {x,y,z} <- R]
```

Generally, we observe that `group` expects as an argument a collection of pairs, and groups them on their first component. If the result needs to be reordered, then this can be done using a comprehension. For example, to group R on its second and third column, and present the results with the original ordering of the tuple components:

```
[{x,y,z} | {y,z,x} <- group [{{y,z},x} | {x,y,z} <- R]]
```

To group a collection and apply an aggregation function to each group, we can use the function `gc`. In general, `gc agFun xs` groups a collection of pairs `xs` on their first component, and then applies the aggregation function `agFun` to their second components. For example, to group R on its first and second columns and return the maximum of the values in the third column for each group:

```
gc max [{{x,y},z} | {x,y,z} <- R]
```

to group R on its second and third column and return the total value of the values in the first column for each group:

```
gc sum [{{y,z},x} | {x,y,z} <- R]
```

to group R on its first column and return the minimum of the values in the third column for each group:

```
gc min [{x,z} | {x,y,z} <- R]
```

to group R on its third column and return the average of the values in the first column for each group:

```
gc avg [{z,x} | {x,y,z} <- R]
```

`gc` is an example of a higher-order function (a function is 'higher-order' if it takes another function as an argument). Two other higher-order functions that IQL supports are `map` and `flatmap`.

`map f xs` applies a function `f` to each member of a collection `xs`. For example, to add 10 to the first component of each tuple of R:

```
map (lambda {x,y,z} {10 + x, y, z}) R
```

We can always use comprehension syntax instead of using `map` e.g.

```
[{10 + x, y, z} | {x,y,z} <- R]
```

More generally, `map f xs` is equivalent to `[f x | x <- xs]`

`flatmap f xs` applies a collection-valued function `f` to each member of a collection `xs` and applies `++` to the resulting collections. For example:

```
flatmap (lambda {x,y,z} [{10 + x, y, z}]) R
```

gives the same result as the previous two queries.

4 Using IQL in AutoMed

Currently IQL is used in AutoMed in two ways: to express the queries within add, delete, extend and contract transformations, and to express queries on schemas defined within AutoMed's Schemas and Transformations Repository (STR). Such schemas may be data source schemas, intermediate schemas or global schemas. In all cases, the individual constructs of a schema are identified by their *scheme* within IQL queries, and they return lists of values[2].

For example, consider a schema construct `<<Student,name>>` representing the attribute `name` of a table `Student`, which has a single-attribute primary key `studentId`, say. The following IQL query counts the number of distinct students' names and sees if this equals the number of student id values:

```
(count (distinct [n | {s,n} <- <<Student,name>>])) = (count <<Student>>)
```

Note that this is how a primary key constraint could be specified using IQL.

As another example, suppose we have in a global relational schema attributes `<<Student,address>>` and `<<Student,age>>` possibly sourced from different databases, and we want to return the set of all known students with all their known address and age information. Here is a query that does this:

```
[{s,a,g} | {s}<-<<Student>>;
           {s,a}<-<<Student,address>>;
```

[2]Generally, the extent of a scheme may also be a bag or a set. Currently, all the AutoMed wrappers return list-valued extents for data source schemes. This may shortly be changing!

```
           {s,g}<-<<Student,age>>]
++
[{s,UnknownAddress,g} | {s}<-<<Student>>;
           {s,g}<-<<Student,age>>;
           not (member [s1 | {s1,a} <- <<Student,address>>] s)]
++
[{s,a,UnknownAge} | {s}<-<<Student>>;
           {s,a}<-<<Student,address>>;
           not (member [s1 | {s1,g} <- <<Student,age>>] s)]
++
[{s,UnknownAddress,UnknownAge} | {s}<-<<Student>>;
           not (member [s1 | {s1,a} <- <<Student,address>>] s);
           not (member [s2 | {s2,g} <- <<Student,age>>] s)]
```

We observe that this query is expressing an outerjoin of `<<Student>>`, `<<Student,address>>` and `<<Student,age>>` over their common `studentId` component.

User-specified identifiers such as `UnknownAge` and `UnknownAddress` above are treated as constants by the IQL evaluator, i.e. they are returned as-is within query results. However, at present there is no type-checking in IQL (either static or dynamic), and so queries such as `1 + UnknownAge` will generate a run-time exception. It is currently up to the programmer to guard against such type-incorrect expressions being passed to the IQL evaluator for evaluation.

A Syntax for IQL End-User Queries

```
query ::=         expr
                 | Let VarToken Equal query In query
                 | query OpToken query
                 | query Equal query
                 | query expr

expr ::=          IntToken
                 | FloatToken
                 | DateTimeToken
                 | StrToken
                 | VarToken
                 | ConsToken
                 | scheme
                 | LSB query Bar quals RSB
                 | LLSB query Bar quals LRSB
                 | SLSB query Bar quals SRSB
                 | BLSB query Bar quals BRSB
                 | LSB RSB
                 | LLSB LRSB
                 | SLSB SRSB
                 | BLSB BRSB
                 | LSB seq RSB
```

```
                  | LCB seq RCB
                  | LRB query RRB
                  | Lambda expr expr
                  | prefix_op

prefix_op ::=       LRB OpToken RRB
                  | LRB Equal RRB

seq ::=             seq Comma query
                  | query

quals ::=           qual SemiColon quals
                  | qual

qual ::=            query
                  | expr LArrow query

scheme ::=          LDAB scheme_seq RDAB

scheme_seq ::=      scheme_element
                  | scheme_seq Comma scheme_element

scheme_element ::= UnderScore
                  | VarToken
                  | StrToken
                  | IntToken
                  | ConsToken
                  | scheme

StrToken =          '([^']|"''")*'

IntToken =          [-]?[0-9]*

FloatToken =        [-]?([0-9]+)(".")([0-9]+)

DateTimeToken =     ("dt '")([1-9])([0-9])([0-9])([0-9])("-")
                    ([0-1])([0-9])("-")([0-3])([0-9])(" ")
                    ([0-2])([0-9])(":")([0-5])([0-9])(":")([0-5])([0-9])("'")

OpToken =           "<>"|"<="|">="|"++"|"--"|"+"|"-"|"*"|"/"|"<"|">"|
                    "div"|"mod"|"and"|"or"

VarToken =          [a-z][A-Za-z0-9_$.]*

ConsToken =         [A-Z][A-Za-z0-9_$.]*

Let = "let"
In = "in"
```

```
Equal = "="
SemiColon = ";"
LArrow = "<-"
Comma = ","
Bar = "|"
LSB = "["
RSB = "]"
LRB = "("
RRB = ")"
LCB = "{"
RCB = "}"
Lambda = "lambda"
LDAB = "<<"
RDAB = ">>"
Colon = ":"
BLSB = "B["
SLSB = "S["
LLSB = "L["
UnderScore = "_"
```

B Built-In IQL Functions

B.1 Unary Functions

```
not      /* (not e) returns the negation of e */
count    /* (count xs) returns the cardinality of the collection xs */
sort     /* (sort xs) sorts the collection xs */
distinct /* (distinct xs) removes duplicates from the list or bag xs */
group    /* (group xs) groups a collection of pairs xs on their first component */
         /* and returns a collection of pairs whose second components are collections */
max      /* (max xs) returns the maximum of a collection of numbers xs */
min      /* (min xs) returns the minimum of xs */
sum      /* (sum xs) returns the sum of xs  */
avg      /* (avg xs) returns the average of xs */
```

B.2 Binary Infix Operators

```
++             /* (xs ++ ys) appends the collections xs and ys */
--             /* (xs -- ys) returns the monus (bag difference) of xs and ys */
union          /* (xs union ys) returns the set-union of xs and ys  */
intersect      /* (xs intersect ys) returns the intersection of xs and ys */
+, -, *, /, div, mod    /* arithmetic operators */
=, !=, >, <, >=, <=, <> /* comparison operators */
and, or                 /* logical operators */
```

B.3 Binary Prefix Functions

```
sub       /* (sub xs ys) returns whether xs is a sub-collection of ys */
member    /* (member xs x) returns whether x is a member of a collection xs */
map       /* (map f xs) applies a function f to each member of a collection xs */
flatmap   /* (flatmap f xs) applies a collection-valued function f to each member */
          /* of a collection xs and then applies ++ to the resulting collections */
gc        /* (gc agFun xs) groups the collection of pairs xs on their first */
          /* component, and then applies the aggregation function agFun */
          /* to the second components of the resulting pairs */
```

B.4 3-ary Function

```
if        /* (if e1 e2 e3) returns e2 if e1 is True, e3 if e1 is False,
             and Null otherwise */
```