The ACeDB Object Oriented Database Query Language.

D. Thierry-Mieg

J. Thierry-Mieg

National Center for Biotechnology Information,
National Library of Medicine, National Institutes of Health,
8600 Rockville Pike, Bethesda MD20894, USA.
E-mail: mieg@ncbi.nlm.nih.gov

Abstract

Acedb is an object oriented database system, originally developed to support the C.elegans then the human genome project and now maintained at the NCBI. It is well adapted to manage and analyze millions of objects with rich but incomplete information and enjoys native support for DNA and proteins. This document presents a renewed query language for acedb, reminiscent of SQL, with a 'select from where' structure, but adapted to the structure of an object oriented schema. Notable features include a semantic alphanumeric ordering such that unc-32 comes before unc-116; strict and loose date comparisons such that 2018-04 is strictly smaller but loosely equal to 2018-04-03; a natural implementation of transitive closure and multivalued data-cells; special support for DNA and Protein queries; and a careful treatment of the extension of Boolean logic to multi-valued variables and missing data. We define the language, show on examples how to construct progressively more complex queries and use shortcuts to facilitate querying the database in interactive sessions. The whole database system is open-source and freely available at ftp://ftp.ncbi.nlm.nih.gov/acedb/ACeDB_NCBI

Contents

1 Introduction

Acedb [1] is an object oriented database which can manipulate complex objects defined as structured trees, with leaves composed of tags, numbers, dates, object references, texts, as well as DNA and protein sequences. The grammar is simple, but powerful, and since the onset of the genome project the system played a central role in part thanks to its powerful graphic interface, and in part because the acedb data structure is very easy to interface with any scripting language, greatly facilitating the work of data curators [2]. A rich example is provided by the AceView web site https://www.aceview.org where an acedb server holds all the data and generates all the graphic displays of an integrative annotation of the human, mouse, rat and worm genomes, extending from the molecular support of each gene in each tissue to phenotypes, diseases and bibliography [6] . The purpose of this document is to present a new version of the query language which is at the same time clear, terse and powerful. While maintaining backwards compatibility, it overcomes the limitations of the previous versions and elegantly captures the particularities of an object oriented database design. As such, parts of the present design could be advantageously applied to other object oriented systems.

2 Presentation of the Query language

2.1 Overview

The first aim of our query language is to allow the systematic exploration of the database by the construction of large tables that can join the data contained in families of objects, subject to textual and arithmetic filtering. In other words, the query language allows to give a relational view of our object oriented database. The syntax of a complex query is reminiscent of SQL, it takes the form

```
1    select x, z from x in ..., y in ..., z in ... where .... order ...
```

In SQL, the variables would iterate through tables. In an object oriented database, the first variable x typically iterates across all the members of a given class, then y is derived *from* x, then z is derived *from* x and/or y. The resulting (x, y, z) tuples are filtered by the *where* condition. Finally, only the (x, z) columns specified in the *select* clause, with lines sorted according to the *order* clause, are exported. The resulting syntax is very natural because in our normal life, we are more accustomed to reason about objects than about relations. For example, the meaning of the rather complex query

```
1  select A, P                          // select 2 variables: A and P
2    from A in class Person,            // A scans the whole class Person
3      P in A->papers                   // P scans the papers of A
4        where count { A->papers } >= 5   // limit to at least 5 papers
```

should not be too difficult to grasp. In this first query, the variable A iterates across all members of class Person, the variable P through all the papers of each author and finally only the authors having at least 5 papers are selected.

Calculations can be performed on numerical values:

```
select p, bmi                    // Report persons with high body mass index
  from p in class Person,        // p scans all persons
    h in p->height,              // h is the height in meters
    w in p->weight,              // w is the weight in kilograms
    bmi = w / h^2                // bodymass index in kg/m^2
      where bmi > 25             // select overweight persons
```

In this second query, using the classical definition of the bodymass index, we select people who are heavy relative to their height. The intermediate variables h and w are not exported, but if they are not available in the database, the *bmi* will not be computed. By combining more variables, and using arithmetic and {} protected embedded subqueries, as in these examples, one can construct large tables which remain relatively easy to develop and maintain. The details on the exploration of the database are explained in section 2, the *where* clause in sections 3, the dates, DNA and protein formats in sections 4, ordering in section 5.

The second aim of the query language is to allow to express very simple things in a very simple way, as in the following examples

```
Find author     // list all members of class author
select K*       // limit to names starting with K
follow papers   // list their papers
select ?Person k* ; >papers     // idem on a single line
select ?sequence kinesin* ; DNA      // export a set of DNA sequences
```

The simplifications (detailed in section 2.5) are obtained in two ways. On the one hand, the queries are implicitly chained, either by being issued in succession, or by being separated by a semi-column. This allow to write only very short query pieces and try them one after the other. On the other hand, the computer is in charge of interpolating the syntactic sugar. For example, the computer expands the second line *select* $K*$ into *select p from p in @ where p like* $'K*'$, where @ (the active set) refers to the results of the previous query. The short form contains all the information but is simpler, easier to grasp, easier to type, and less prone to clerical mistakes.

The details of the implementatin of the query language within the acedb system are described in the annex. The lists of all the keywords is presented in A.1. The history of the development is summarized in A.2. The way to download, compile and test the latest version of acedb is explained in A.3. A complete interactive query session is detailled in A.4. Finally annex A.5 presents the C-language API used in client programs.

2.2 Introduction to the acedb schema

In acedb, each object belongs to a class, with a known schema defined in the file wspec/models.wrm. A simple schema for a bibliography database can be

```
?Paper  Title UNIQUE ?Text            // Class Text is fast searchable
        Author ?Person XREF Papers    // by default, tags are multivalued
        Journal UNIQUE Text           // except if UNIQUE is specified
        Submitted UNIQUE DateType     // Dates have a special format
        Published UNIQUE DateType     //
        Pages Int UNIQUE Int          // several linked values are allowed

?Person Papers ?Paper XREF Author     // XREF maintains the reverse relation
        Affiliation  Text             // Multivalued by default
        Professor                     // Boolean tag, present or absent
```

A few instances of these classes can be

```
Person Tom
  Professor
  Papers p1  // Tom has published several papers
         p2
  Affiliation Paris
              Tokyo

Person Jim
  Papers p1
  Affiliation London

Paper p1
  Title "Quantum computers"
  Author Tom
         Jim
  Journal "Nature"
  Published 2001-04
  Submitted 2000-11
  Pages 1   23
        100 107

Paper p2
  Author Jim
  Published 2004-06
  Pages 17 19
```

A simple biometric and filiation schema can be specified as

```
?Person Filiation Parent  ?Person XREF Child
```

```
                Brother ?Person XREF Brother
                Child   ?Person XREF Parent

        Biometrics   Height UNIQUE Float          // Height in meters
                     Weight UNIQUE Float          // Weight in kilograms
```

The cross referencings (XREF) in the schema guarantee that the author/paper, the parent/child and the brother/brother relations remain synchronized.

2.3 Navigating through the acedb schema

As explained previously, one can list all existing papers using

```
select ?Paper
```

resulting in a 1 column table

```
p1
p2
```

One can also directly access a single object using

```
select OBJECT('Paper', 'p1')
```

We can get to a tag using #, and get the value behind a tag using −>. For example the following query

```
select p in ?Paper, a in p->Author,prof in a#Professor, z in a->Affiliation
```

will produce the 4 column table

```
p1  Jim      NULL          London
p1  Tom  Professor   Paris
p1  Tom  Professor   Tokyo
p2  Jim      NULL          London
```

Reporting the pages creates a new challenge. According to the schema, the pages tag supports a two columns table of integers. To access the two columns, one must use square brackets.

```
select P in ?Paper, p1 in P->pages, p2 in p1[1] where p2
```

or the equivalent query, in which p sits on the pages tag of the object

```
select P,p1,p2 from P in ?Paper,p in P#pages,p1 in p[1],p2 in p[2] where p2
```

or the equivalent query, in which the bracket is associated directly to the tag name

```
select P,p1,p2 from P in ?Paper,p1 in P->pages[1],p2 in P->Pages[2] where p2
```

In each case, one obtains

```
p1 1   23
p1 100 107
p2 17  19
```

Several details in the second form of the query should be noticed. a) The variable names are case sensitive, p and P are distinct. b) In principle $p[2]$ should iterate through each number in column 2 behind p

```
select P,'last page',p2 from P in ?Paper,p in P#pages,p2 in p[2] where p2
```

gives

```
   p1 last page 23
   p1 last page 107
   p2 last page 19
```

but when we have 2 bracketings of the same variable, they remain synchronized so $p[1]$ and $p[2]$ always refer to the same line of the table behind the tag *pages*. This synchronization is very important when a tag is multivalued. Consider a schema where each chromosome contains the position of all its genes

```

Chromosome chr_18
Gene A  1000 2000
Gene B  4000 6000
```

The 3 equivalent queries

```
select c,g,a1,a2 from c in ?chromosome,g in c->Gene,a1 in g[1],a2 in g[2]
select c,g,a1,a2 from c in ?chromosome,g in c->Gene,a1 in g[1],a2 in a1[1]
select c in ?chromosome, g in c->Gene, a1 in g[1], a2 in a1[1]
```

gives the expected answer, with one line per gene, with the correct coordinates

```
   chr_18  A 1000 2000
   chr_18  B 4000 6000
```

because on each line the coordinates $a1$ and $a2$ are derived from, and therefore are associated to, the gene g. But if we write the following more complex query where we reinitialize the definition of the 2 coordinates $a1$, $a2$ at the root of the object

```
select c, g, a1, a2
       from c in ?chromosome,
       g in c->Gene, a1 in c->Gene[2], a2 in c->Gene[3]
```

$a1$, $a2$ are no longer associated to the variable g and we obtain the silly combinatorial answer

```
chr_18  A 1000 2000
chr_18  A 1000 6000
chr_18  A 4000 2000
chr_18  A 4000 6000
chr_18  B 1000 2000
chr_18  B 1000 6000
chr_18  B 4000 2000
chr_18  B 4000 6000
```

2.4 Sum, min, max and average

The MIN, MAX, SUM or AVERAGE of any numerical variable, for example a multi-valued numerical field, can be evaluated by interpolating in capitals the desired operator in front of its definition

```
select p,x from p in ?Paper , x in MAX p->pages
select cc,x from cc in ?chromosome, x in AVERAGE cc->gene[2]
```

Rather than giving in x the list of all values associated to the query, one per line, one obtains for each object the required result as a single value.

2.5 Multi-valued data cells and transitive closure

In an object oriented database, tags are often multivalued. As already explained, the expand operator $->$ gives access to the values behind a tag, one value per line. But in a table with several columns, if several tags are developed, this leads to a combinatorial number of lines. Suppose *tag*1 has 4 values and *tag*2 has 6 values, the query *select* @$->$ *tag*1, @$->$ *tag*2 would give 24 lines. To limit this proliferation, it is sometimes more convenient to export all the values in a single data cell. The parallel-expand operator => provides this possibility.

```
    // => gives the parallel expansion, unsorted
    // The results appear in the same order as in the object
    // as desired when listing the authors of a paper
select ?paper p1; =>author
  Tom; Jim // all authors come on a single line

    // Genealogy example
select m,'is the mother of', m=>child from m in ?Person where m == eva
  Eva is the mother of Cain; Abel; Seth
```

Another modification is the transitive-expand operator >> which gives the transitive closure of a tag, meaning that the >> operator is used iteratively. For example *grand_ma* $->$ *child* lists

all the direct children of grand_ma, but *grand_ma >> child* gives her children, grand-children, great-grand-children ... For example, if the first two biblical generations are known in the database

```
select p, 'is a parent of', from p  in ?Person, c in p->child where c
   Enosh is a parent of Baraki
   Enosh is a parent of Keman
   Eva   is a parent of Abel
   Eva   is a parent of Cain
   Eva   is a parent of Seth
   Seth  is a parent of Enosh
```

The whole progeny of Eva is given by the *child* transitive closure

```
select ?Person Eva ; @ >> child // whole progeny
  Abel
  Baraki
  Cain
  Enosh
  Keman
  Seth
```

2.6 Simplification rules

The most frequent way to define the first variable is to iterate through a class. To list all members of the class sequence known in the database, one may write

```
select x from x in class "sequence"  // full syntax, or equivalently
select x in class "sequence"  // drop from since x is exported
select x in class  sequence   // drop the quotes
select class sequence         // drop the name of the variable
select ?sequence              // use ? as a shortcut for class
find sequence                 // intuitive equivalent syntax
```

The successive short cuts are all acceptable, because the original syntactic sugar was not bringing any information. First, the purpose of the *select ... from ...* syntax is to export only some of the arguments that are needed to compute, or to order, the exported columns in a specific way. If all columns should be exported the '.... *from* ' clause is no longer needed. Then, the doubles quotes around the class names are optional, as class names in acedb never contain spaces or non alphanumeric characters. The variable name x is redundant since x was not reused. The very frequent keyword *class* can be abbreviated as a leading question mark. Finally *find*, at the beginning of a query, is used as an alias of *select class*. The general idea is to allow the most concise syntax and remap it into a full query which is executed by the database.

Suppose we want to find all the members of *class Person* called *King*. Again this name does not contain any blank space, so starting from the full syntax, we can write any of the following equivalent queries

```
select x from x in class "Person"  where x like "king"  // full syntax
select ?Person like "king"   // drop x  and the where keyword
```

When the variable is not named, everything to the right of the class name is interpreted as being part of the *where* clause. One can replace the *like* by ~ (tilde), or replace it by one of the usual comparators < <= == > >

```
select ?Person < kj          // select Abel to King, but not Kong
```

Finally, if there is nothing else in the query we can eliminate the operator symbol

```
select ?Person King    // which is interpreted as
select a in ?Person where ( a#King  or a like "King" )
```

we select at the same time the persons named 'King' or containing the tag *King*, In most situations, this is not ambiguous but using this simplified syntax to select a person with the meaningful name King may also select the current *King* of the kingdom !

The double quotes are not always needed. Indeed the comparison operators expect a declared variable, a number or a string on the right hand side. We do need to protect strings containing spaces or special characters, like slash or minus. We must also protect strings matching the reserved word, i.e. "select", "from" or strings matching the name of the declared variables, or tag names like in the previous example distinguishing the name "*king*" and the tag *King*. Attention, variable names are case sensitive, x and X are distinct, but reserved words (*select*, *Select*, *SELECT*) are not. Consider these 2 examples

```
select x in ?Person, y in x->Brother where y > x    // x is a variable
select x in ?Person, y in x->Brother where y > "x"  // "x" is letter x
```

The first query will select pairs of persons (x, y) where the name of *brother* y if alphabetically behind the name of *brother* x, for example (Abel, Cain), but not (Cain, Abel). Whereas the second query requests that y starts with x, y or z and would accept (Zachary, Xavier), but reject (Abel, Seth).

There is a special issue concerning the naming of the instances of a class. By default the instance names are not case sensitive, but the firt typography encountered when parsing the data is always preserved. For example if in the datafiles the first spelling is "*Eva*", then the query *Find person eva* will return *Eva*. However, one may specify in acedb that the instance names of a particular class are case sentitive, this is specified in the self documnented configuration file *wspec/options.wrm*. This option is used for example in the class *Gene* for the model organism *Drosophila melanogaster* which, according to tradition, has two distinct genes, one called *A* and the other called *a*.

Finallly, on the command line, the leading keyword *select* can be abbreviated as *s*,

```
select ?Person  k*           // full form
s ?Person  k*                // s   is a short form for select
```

All these shortcuts allow to type quickly terse queries on the command line interface, which are first mapped in a well defined way into a full fledged query language, then evaluated.

2.7 Chaining queries using the active list

The second most frequent way to derive the first variable of a query is to iterate through the active list. This concept was inspired by the Unix pipe. The active list is called @. It is a mathematical set, always sorted, and with no duplicates. It starts empty. It is then populated or modified by each successive *select* statement. This allows to chain the queries like in:

```
1     select a from a in class "Person"          // populate the active list @
2     select a from a in @ where a like "king"   // derive a from the list
3
4        // which can be simplified into:
5     find person
6     select king
```

If a query exports several columns, the active list corresponds to the first exported column. For example

```
1     select p, j from p in class paper, j in p->journal  // case PJ
2     select ?paper, >journal     // equivalent short form of case PJ
3     select j, p from p in class paper, j in p->journal  // case JP
4     select j    from p in class paper, j in p->journal  // case J
```

Case PJ will export a list of (paper, journal) tuples and the active list contains papers. Whereas case JP will export (journal, paper) tuples and the active list will contain journals, and case J will only export journals and the active list will also contain journals.

2.8 Silent queries

In many situations, one may want to run a query to know the number of objects satisfying the query without wishing to see the resulting table. As in mathematica and other interactive languages, the output can be suppressed simply by adding a semi-column at the end of the query.

```
1     select ?Person ;   // silent query endding with semi-column
```

Capitalizing on the concept of an active list and on the syntax shortcuts explained in the two previous sections, the semi-column can be used to chain queries:

```
1     select ?Person ;   // all persons, silent
2     select Tom ;       // restrict to Tom, silent
3     select >papers     // export papers of author Tom
```

The result of the first query, in this case the set of all persons, is not exported, but is used as input to the second query, which selects author Tom, then to the third query. In fine this chained query exports the list of papers authored by Tom.

The same query can be presented on a single line:

```
select ?Person ; tom ; >papers // export Tom's papers
```

3 Filtering on arithmetic and textual conditions

3.1 Comparators

The *where* clause is used to filter the results. The simplest form is to search for the specific name of an object

```
select ?Person Tim
```

which expands as

```
select p in class "Person" where p == "Tim" // select a single person
```

The usual comparators: < <= == >= > can act on names, dates and numbers. In addition, a numeric variable is interpreted as a real number and numerical comparisons can involve arithmetic: additions, subtractions, multiplications, divisions, modulo, power. Parenthesis are recognized in the usual way:

```
select p in ?Person,
  h in p->height,                    // assume h is given in meters
  w in p->weight                     // assume w is given in kilograms
    where 2 * (h - 100) < 1.7 * w  // select relatively heavy people
```

In text comparisons, the value of a variable is its name, with the caveat that acedb interprets embedded numbers as numbers, implying the ordering

```
a < a7b < a11c < b
```

String matching comes in 2 flavors. Using the syntax a like b, or equivalently a ~ b, one can invoke a simple system with 2 kinds of jokers: question mark (?) to represent a single character or star (*) to represent an arbitrary string.

```
p ~ "T?m"      // selects Tam, Tim and Tom,
               // but not Attim which does not start with T
p ~ "T*m"      // also selects Theotym,
               // but not Thomas which does not end with m
p ~ "*T?m*"    // also selects Attim"
p ~ "*T*m*"    // selects Tam,  Tim, Tom, Theotym, Thomas and  Attim
```

Single quotes prevents the expansion of the (*) and (?) symbols

```
p == '*'        // select person actually named *
```

3.2 Regular expressions

Alternatively, one may write a equal-tilde b to invoke the full C regular expression matching

```
p =~ 't[io]m'     // selects Tim, Tom and Attim but not Tam
p =~ '^t[io]m'    // selects Tim and Tom,
                  // but not Attim which does not start with T
p =~ 't.*m'       // selects Tam,  Tim, Tom, Theotym, Thomas and  Attim
p =~ 't.*m$'      // rejects Thomas which does not ends with m
```

The regular expressions must be protected by single quotes to prevent premature evaluation.

3.3 Automatic classification using dynamic subclasses

In most situations, the class of a variable is known from the schema. Requesting $paper->author$ would be known in our examples as yielding an author, i.e. an instance of *class Person*. However, acedb allows dynamic classification into sub-classes. For example, one can define in acedb a subclass $Prolific\_author$ as all authors of at least 5 papers known in the database. Whenever an *Person* exceeds this threshold, it belongs at the same time to the *class Person* and to the subclass Prolific_author. To get the full list of prolific authors one may directly list the subclass. But it may be necessary to check the status of a given set of authors, this is done using the operator ISA (is an instance of a class) as follows

```
  // if Prolific_author is a subclass of class Person one may
select PA in class Prolific_author       // list the subclass
                  // or select objects belonging to a subclass
select A ...          // select persons to populate the @ list
select PA from PA in @ where PA ISA Prolific_author // filter
```

The subsclasses are defined in acedb in the configuration file *wspec/subclasses.wrm*

3.4 Combining Boolean conditions: AND, OR, XOR and NOT

As in nearly all query systems, conditions can be combined using (OR, XOR, AND, NOT) in that order of precedence and parentheses. The names of the operators are not case sensitive: (XOR, Xor, xor) are equivalent. (OR, XOR, AND, NOT) can be abbreviated as (|| ^^ && !). Remember that the single caret is reserved for arithmetic power 2^3 = 8. For example, one may write

```
... where (a <= x && x <= b) || (b < x && x < a)
```

to specify that x is either in the segment [a,b] or in the segment]b,a[. However, as discussed below, missing data and multivalued variables slightly complicate the semantics of the Boolean operators.

3.5 Dealing with missing data: TRUE, FALSE, NULL and NaN

In a standard programming language, like C, it is standard practice to assume that the numerical value zero matches the Boolean value FALSE. For example a looping instruction $while(x)$ is equivalent to $while(x\ !=\ 0)$. A standard error, however is to evaluate a non initialized value, but this can be detected at compile time and fixed before running the program. Apart from zero, non-zero and non initialized values, a 4th type of number occurs in a program, the infamous NaN (Not A Number), which is generated as the results of an invalid arithmetic operation like $1/0$ or $log(0)$. Unfortunately, these are only detected at run time, and may occur very frequently for example in *Deep Learning* programs (a branch of artificial intelligence).

In a database query, it is the rule, rather than the exception, to stumble upon a non initialized value. Indeed, all fields start empty. For this reason, the logical tests in the query language are not binary but ternary, the 3 allowed values being TRUE, FALSE and NULL.

Boolean tags, $p\#tag$ evaluate as NULL if p is not defined, otherwise as TRUE if the object p contains the tag, or FALSE if it does not.

Values, e.g. $(p{-}{>}weight)$, evaluate as NULL if undefined, i.e. if the object p is not defined, or if the tag $weight$ is not present in the object, or if the tag does not point to a value in the object. Otherwise they evaluate as TRUE even if the value happens to be zero.

In numerical calculations, the NaN value is sticky. Any calculation involving one NaN evalutes to NaN, for example $log(0) + 4$. Similarly, any calculation involving a missing data evaluates to NULL, for example $(p\ {-}{>}\ weight + p\ {-}{>}\ height)$ evaluate as NULL if either $weight$ or $height$ is not specified in the object p.

Any string comparison involving a NULL value (missing data) and any number comparison involving a NaN or a NULL evaluates to FALSE.

Count operations always evaluate to a valid number.

In Boolean operations (AND, OR, XOR, NOT), NULL and NaN evaluate as FALSE.

This may seem a little abstract, but is easy to understand on examples, and important because, in a database query, undocumented values are very frequent.

```
1  select A from A in ?Person where A                    //   Always true
2  select A from A in ?Person where A#Professor     //  Tom is a Professor
3  select A from A in ?Person where A->Professor    //     False, no value
4  select P from P in ?Person where P->weight       //    Is there a value
5  select P from P in ?Person where P->weight > 80  //     Check the value
6  select  1,x from x = (-1)^0.5 where x            // False: not a number
7  select  1,x from x = (-1)^0.5 where 1 OR x       //  True: T or NaN = T
```

Combining the filters, one can for example quality check the data and export all cases where the values are either missing or out of range, for example a weight shoud never be negative:

```
select p,w from p in ?Person, w in p->weight where w <= 0 OR NOT w
```

3.6 The meaning of NOT in a multivalued universe

In standard arithmetic, the two conditions (a != b) and !(a == b), i.e. (a NOT EQUAL b) versus (NOT (a EQUAL b)), are equivalent, they are FALSE if (a == b) and TRUE otherwise. But in the context of a database query, the situation is more involved. In general, the variables are multivalued, and the meaning of NOT depends on a choice of strategy. Consider the case of two papers, $p1$ with authors Tom and Jim, and $p2$ just with author Tom. We certainly expect Paper $p1$ to answer TRUE to the query ($author$ == Tom). Therefore it should answer FALSE to the query (!($Author$ == Tom)). But what about ($Author$!= Tom). Scanning through the author list of $p1$ we get 2 answers {TRUE, FALSE} and we need a strategy to reduce this list to a single answer. We chose to favor TRUE, i.e. the OR value of all individual answers. Therefore $p1$ (but not $p2$) answers TRUE to ($Author$!= Tom) because $p1$ has at least one other author.

In conclusion, if the variables are multi-valued, the following queries are not equivalent

```
select p from p in class Paper where p->author != Tom
select p from p in class Paper where NOT p->author == Tom
```

With our choice of strategy: TRUE wins over FALSE. So in the first case, one find all papers where Tom is not the only author in the second case one finds all papers where Tom is not an author,. In addition, according to the previous section discussing missing data, the papers without a known list of authors are also selected in the second query, but not in the first query.

3.7 Counting elements in embedded subqueries

A very useful constraint is counting. For example, prolific authors can be found by either by counting the number of values of a tag, or the number of lines of an embedded subquery delimited by curly brackets {}, i.e. they can be found by the 2 equivalent queries

```
select a in ?Person where count a->papers > 5
select a in class author where count {select p in a->papers} >= 5
```

but in the latter case, we can become very specific and only count papers published in a given journal

```
select a in class "Person" where
  count {select p in a->papers where p->journal == "nature"} >= 2
```

4 Basic data types

4.1 Constants and numerical calculations

A constant is declared using the equal symbol. The constant may be a number of a string. These constants can then be used in calculations and filters

```
select x, y, z from x = 2 , y = "hello", z = 2 * 3 + 4
select L from L in class line , pi = 3.14 where L->length > 2 * pi / 3

select d from  d = 2 where d == 5 - 3   // correct filter
select d from  d = 2 where d = 5 - 3    // error, please use ==
```

As usual in C and many other languages, setting the value of a constant uses the = sign, but assessing an equality in a *where* clause uses the == symbol. Looking closely, we may notice that since all the variables x, y, z, L, d are exported, the *select* - *from* is redundant, moreover the x variable is never reused, so its declaration is redundant. This lead us to the simplified forms:

```
select x from x = 2 // full form
select x = 2        // short form
select 2            // shortest form
```

Since the system understands the parenthesis and the standard arithmetic operators (plus, minus, multiply, divide, power, modulo) we actually have a (multi dimensional) calculator

```
select  (1 + 2) * (3 - ( 4 + 5))  // returns  (3) * (-6) = -18
select  8 modulo 3                // returns 2
select  -1 modulo 3               // returns  2 (math convention)
                                  // rather than  -1
select  2 * 5 modulo 3            // returns  1
select  9 ^ 2                     // returns square(9) = 9 * 9 = 81
select  9 ^ (1/2)                 // returns sqrt(9) = 3
select  (-1) ^ .5                 // sqrt(-1)returns NULL, arithmetic exception
select  3*2 , (5+1)/2             // returns 6,3
```

When computing -1 *modulo* 3, we prefer the mathematical convention 2 because the C language convention -1 breaks the periodicity of the modulo function and introduces complications when translating DNA codon triplets. All numerical calculations are performed using 64-bits floating numbers (C doubles).

4.2 Loose and strict date comparisons, date differences

Dates are special, because they may be specified with a very variable precision, sometimes just a year is provided, sometimes down to the second. To allow more meaningful comparisons we introduce two kinds of comparators, using either the lower or the higher precision.

A constant date is declared by enclosing the numbers using the back-quote sign:

```
select d = `2016-01-30_22:47:15`
```

The successive numbers represent Y:year, M:month (1 to 12), D:day (1 to 31), h:hour (0 to 24), n:minute (0 to 60) and s:seconds (0 to 60).

In a user provided date, the leading zeroes may be dropped, they are reinstated in the answer:

```
query::   select d = `2016-1-17_3:7`   // No leading zeroes
answer::  2016-01-17_03:07             // standardized date format
query::   select d = `20160321`     // No minus signs
answer::  2016-03-21                // standardized date format
```

In a database, the objects are usually dated with a certain granularity. For example the submission date of papers often gives the day, but for books, we often only know the year. Also a frequent query would be to retrieve the papers of a given year or a given month. To satisfy these needs, we introduce 2 kinds of date comparisons, strict and loose. In a strict comparison, using the comparators <= == >= the 2 dates are compared at the highest available precision, just like one would compare real numbers. In the loose mode, using the tilde comparators <~ , ~ , =~ , >~ , the 2 dates are compared at the lowest precision, if one date is given in years, the other is rounded in years. As a result, we have

```
    `2016-2`  == `2016`       // ---> false
    `2016-2`  >= `2016`       // ---> true
    `2016-2`  <= `2016`       // ---> false

    `2016-2`  =~ `2016`       // ---> true
    `2016-2`  >~ `2016`       // ---> true
    `2016-2`  <~ `2016`       // ---> true

    `2016-2`  == `2016-2-17`  // ---> false
    `2016-2`  == `2016-3`     // ---> false

    `2016-2`  =~ `2016-2-17`  // ---> true
    `2016-2`  =~ `2016-3`     // ---> false
```

Notice that a date must always be protected by a pair of back-quotes. This need is obvious if you want to specify the month, since the date `2016-3`, meaning march 2016, is not the same thing as the subtraction $2016 - 3 = 2013$, but it is also needed even when you just give a year like `2013` to trigger the interpretation of 2013 as a date.

Finally, like in SQL, it is possible to compute a date difference at a desired pecision using DATEDIFF (unit, date_1, date_2) where unit can be one of (year, month, day, hour, minute, second), or abbreviated as (y,m,d,h,n,s), notice the n for miNute. The result is an integer number.

In all cases, be very careful when using dates and always check the validity of the query on a few known cases since it is very easy, when dates are involved, to mean something and write something else.

4.3 DNA and proteins, motif searches

A specificity of acedb is that it understands the genetic code. If the database contains sequences, or mRNAs, or any object with a DNA tag, one can obtain its DNA using

```
   select s in class sequence, d in DNA(s)
// yields the full DNA of each sequence in the database.
// To obtain a specific part use
   select x = 3, y = 8, s in ?sequence, d in DNA(s,x,y)
// The reserved word DNA must be written in upper case, this
// allows to use dna as a variable name or to access
// the DNA tag. If x > y, one obtains the reverse complement
   select s, dna, adn, from s in ?sequence,
      dna in DNA(s,1,6), adn in DNA(s,6,1)
// gives
   my_sequence  atgttg    caacat
// PEPTIDE will translate the sequence
   select PEPTIDE (@,1,2)
// gives Methionine-Leucine (atg codes for Met, ttg for Leu)
   ML
// To get all proteins encoded in messenger RNA, try
   select ?sequence CDS      // the CDS tag means the sequence is coding
   select -o f ?sequence CDS ; PEPTIDE // all proteins are written to file f
```

Notice that acedb will translate nuclear DNA using the standard genetic code, but if the relevant standardized information has been provided in the database, it will translate some sequences using a different genetic code, for example the human mitochondrial code to translate human mitochondrial mRNAS.

In some cases, one may have the name of a chromosome, say '12', exists in 2 different classes, say class 'Map' and class 'Sequence', or simply is a number, we can 'reclass' the variable 'm' using OBJECT before extracting the DNA:

```
select x,s,dna from x=11+1, s in OBJECT('Sequence',x), dna in DNA(s,101,112)
select m,s,dna from m in ?Map where m == '12',
       s in OBJECT('Sequence',m), dna in DNA(s,101,112)
```

In principle, one can select all DNA sequences coding for a particulat motif or coding for a particular peptide motif say MLR or MIR (Methionine (Iso)Leucine aRginine) using simple text comparison (like) of full UNIX regular expressions (equal tide)

```
  select s,dna from s in class sequence, dna in DNA(s) where dna like '*taag*'
  select s,p from s in class sequence, p in PEPTIDE(s) where p  =~ '^M[IL]R'
```

However, the content of the sequences is not indexed, and dedicated tools like BLAT or BLAST would be much faster for large scale searches of multiple motifs.

5 Column titles and row order

The result of a query is a tab delimited table. The order of the columns is specified by the order of the variables in the *select* clause. If the option $-t$ is provided, the first line, prefixed by a #, will contain the title of the columns. By default, the title of the columns are the names of the variables, but a more complete title can be provided for some columns using the $TITLE$ field (in capitals) as in:

```
select -t x, y, z from ... where ... TITLE x:this is x, z:this is z
```

Using the expanded options $-o$ file_name $-title$ my_title, as in

```
select -o foobar.txt --title ''Hello World''
  x, y, z from ... where ... TITLE x:this is x, z:this is z
```

adds at the top of the output file 'foobar.txt' a first line prefixed by ### with the current date, the chosen title, and the name of the output file.

The order of the rows is by default the alpha-numeric ordering. But this order can be modified using the following syntax.

```
    // By default the lines are sorted alphanumerically
select x, y, z from ... where ...                    // default order

    // The order can be modified using order_by
select x, y, z from ... where ... order_by 1+2+3  // default order
    // to order by y increasing, then x decreasing, then z increasing
    // these 2 constructions are equivalent
select x, y, z from ... where ... order_by 2-1+3  // use column numbers
select x, y, z from ... where ... order_by y-x+z  // use variable names
```

In the leading column, or the following ones in case of a tie, numbers are sorted numerically : (-5 -2.7 -2.6 0 3 7 11 11.1 12) and names are sorted alphabetically. However, we found [1] that it is much nicer to order names containing embedded number numerically as in (a9 b7 b7.1a b11). The rule is to order numerically on successive groups on contiguous numbers. Notice that this method also works well for times and dates '9:20' will come before '10:2'.

6 Conclusion

As from 2018, the official site for the acedb object oriented database system has been transfered from the Sanger centre to the NCBI. We presented here the new query language developped to allow a faster and more fluid access to the data, while maintaining a good compatibility with the previous systems.

The main idea was to clearly define a full fledge syntax while at the same time developing a set of simplification rules which allow in the simple cases to write very terse queries and let the computer

interpolate the syntactic sugar. This double strategy allows an integrated support for easy to read verbose scripts and for easy to type terse interactive commands. All queries are analyzed by the same machinery which interprets the query, checks the syntax, and executes the search in C while taking advantage of current state of the acedb data caches.

Some aspects of this presentation may be interesting to a broader audience not necesseraly using the acedb system. For example the implicit handling of multivalued fields is much simpler and more natural than in relational databases. The ternary logic: True, False, Null presented in section 3.5 is important in any situation where the information is often incomplete. The distinction between 'A not equal B' and 'not A equal B', detailled in section 3.6, is critical whenever the variables are multivalued. The native support for DNA and proteins is handy in biological applications. Finally. the strict and loose date comparisons of section 4.2 can be appreciated as the cherry on the cake.

Acknowledgments

We would like to thank Sam Cartinhour, Michel Potdevin, Mark Sienkiewicz and Richard Durbin for many discussion on the best way to query an object oriented database and all acedb users for a continuous flow of interesting complex queries with unexpected answers. This research was supported by the Intramural Research Program of the National Library of Medicine, National Institute of Heath.

References

[1] Richard Durbin and Jean Thierry-Mieg The acedb genome database, Computational Methods In Genome Research, pages 45-55. Edited by S. Suhai, Plenum Press, New York, 1994

[2] Richard Durbin and Jean Thierry-Mieg Syntactic Definitions for the ACEDB Database Manager Technical Report: MRC Lab. for Molecular Biology, 1992

[3] Lincoln D Stein, Jean Thierry-Mieg Scriptable access to the Caenorhabditis elegans genome sequence and other ACEDB databases Genome research, 8,12 pp 1308-1315, 1998

[4] Jean Thierry-Mieg, Danielle Thierry-Mieg and Lincoln Stein ACEDB: The Ace Database Manager In Databases and Systems, 2001 DOI: 10.1007/0-306-46903-0_23

[5] Sulston...the C.elegans genome

[6] Danielle Thierry-Mieg and Jean Thierry-Mieg AceView: a comprehensive cDNA-supported gene and transcripts annotation. Genome Biology 2006, 7(Suppl 1):S12.

[7] Danielle Thierry-Mieg and Jean Thierry-Mieg Magic RNAseq pipeline

[8] Code and documentation are available from *https : //www.aceview.org/Software/ https : //github.com/ncbi/AceView https : //github.com/jtmieg/AceView*

A Annex

A.1 List of operators and reserved keywords

To summarize, the full list of operators, in their order of precedence is

```
1   ; order\_by
2   from select , where
3   in =
4   || OR ^^ XOR &&  AND ! NOT
5   like =~ ~ == != >= <= > < >~ <~
6   ISA
7   + - * / ^ modulo
8   class
9   DNA PEPTIDE DATEDIFF
10  count
11  >> -> # => :
12  @
13  object
14  () {} []
15  BACKQUOTE QUOTE DOUBLEQUOTE
```

with the restriction that the 3 kinds of parentheses and also the quotes must be nested correctly (["bb"]) is correct but (x[y)] is illegal.

A.2 History of the development of the acedb query language

The query syntax described in this document results from the convergence of several developments - the original acedb query language of 1990, available in the 'query' box of the acedb graphic interface - the table maker system of 1992, with its user friendly graphic interface - the original AQL query language developed at the Sanger around 2002

The general idea was to learn from the 3 existing systems, conserve the best aspects of each, fix their limitations and clean up the interface. For example,

– The acedb query language is very terse and expressive, but it is limited to constructing sets of objects and does not allow to display the content of selected fields.

– The table maker was meant to fix these limitations. Using its rich graphic interface, one can easily construct a tables involving objects, numbers, texts and even DNA sequences. However, even a simple tables cannot be constructed on the command line. One need always to construct the table graphically, to save its definition in a definition file, and run the query using that file.

– The original AQL query language was ment to fix this limitation. AQL defined a command line syntax, reminiscent of SQL, but adapted to the idiosyncrasies of an object oriented database like acedb. It made it in principle possible to compose a table on the command line or using a library call. Unfortunately, the syntax was slightly too rich, encouraging the user to write convoluted queries, and always verbose. Since there was no graphic interface to help compose a valid query, and since the original AQL compiler did not explain the eventual errors, it was hard to write a non-trivial valid query. Finally, AQL execution was slower than table-maker, and lacked access to DNA and protein sequences.

From the user point of view, the previous interfaces are maintained, but the old style queries are translated internally and executed by the new query engine. In particular, the graphic table-maker interface remains valid and can still be used to construct complex queries and most original AQL queries remain valid. In practice the shortcuts defined in section 2.5 map the terse query language of the acedb graphic interface, first released in 1990, into a (modification of) the full fledged AQL queries developed around 2002. As a result, these 2 types of queries, and the table constructed via the acedb graphic table maker interface, are all expressed in the new unified syntax and the database exploration can be executed using a single highly optimized query engine.

A.3 Code availability

The acedb source code can be downloaded from

https://github.com/ncbi/AceView

The tar ball is effectively identical to the distribution of the AceView/Magic pipeline [7] which uses acedb as the underlying database, and can organize the workflow and manage the meta-data of tens of thousands of next-generation sequencing runs. The same package underlies the Gene annotation public web site https://www.ncbi.nlm.nih.gov/AceView [6]

The program is written in C and has very few external dependencies. To compile the code try

```
1  tcsh
2  gunzip -c magic.*.tar.gz | tar xf -
3  setenv ACEDB_MACHINE LINUX_4_OPT
4  cd magic*
5  make libs tace
6  make -k all
```

The code can also compile on the Mac using setenv ACEDB_MACHINE MAC_X_64_OPT or on any other linux/unix platform. Multiple choices are offered in the directory wmake. The tace command-line interface presented in the present document should compile without difficulty. The xace graphic-interface requires the installation of the public X11 development environment as explained in the README file situated at the top of the acedb distribution.

A test-bench containing examples of queries can be constructed using the command

```
1  tcsh wbql/bqltest.tcsh
```

After running the script, the file ACEDB_QUERY_LANGUAGE_TEST/test.out contains the output of the commands contained in test.cmd and the file test.diff, which contains the differences between test.out and test.out.expected, should be empty. The database can be examined and further queries can be tested using the command

```
bin*/tace ACEDB_QUERY_LANGUAGE_TEST
```

which will activate the command line interface of acedb. All the examples in the present document can be copied in the interface and should give proper answers. A question mark will invoke the on-line help and list all the possible commands. In particuler, the acedb command 'show' will display the full content of the active objects and will help to understand why they where selected by the query.

Please try the code and send us feedback.

A.4 Running an acedb session

As explained above, a small test database can be constructed automatically. More generally, the directory waligner/metadata contains some examples of large acedb schemas. Once you have constructed a local directory containing an empty sub-directory database and a sub-directory wspec defining the schema and a data file xxx.ace, you can initialize a database and parse an ace file as follows:

```
  bin*/tace .  // launch the compiled text-acebd on the local directory
y              // yes, initialize the database
   parse myfile.ace              // parse some data
   save  // save the data in compiled binary format
   quit
```

Acedb can be regarded as a data compiler. It can read successively several data files, merge them and compile them into a compact binary format allowing fast retrieval. Clear messages signal places in the data file which do not conform to the schema. You can fix the file and reload it. Reading the same file several times is idempotent, i.e. it does not cause any problem.

Subclasses are configured in the file wspec/subclasses.wm

```
   Class Prolific_author      // The class name must start with a letter
   Visible                    // Make it available in graphic menus
   Is_a_subclass_of Person    // Inherit from class Person
   Filter "COUNT Papers >= 5" // Chosen filter
```

One can then use the database and ask queries on the command line or import it from a file using the $-i$ parameter or redirect the output to a file using $-o$

```
  bin*/tace . // start the system
    select ...  from ... where // a query
xx yy zz      // the reply comes on stdout
```

```
xx yy zz
    select -o f.out ... from ... where ...
                // the reply goes in the file f.out
    select -a ... // the reply is quote protected (.ace format)
    select -i f.bql ... // file f.bql contains the query
    select -i f.bql -s  // silent mode
    select @ ... // use the new active set
    ? select  // help of the command line parameters
    ?         // list all the acedb commands
    quit
```

In silent mode (with semi column at the end of the query, or using the option -s (-silent)) the output is suppressed but the active list is modified.

Acedb contains dozens of commands available by typing ? or ?command. There is also a rich graphic interface, available using the command *bin.* $*$ */xace.*, which generates on the fly in HTML5/SVG language all the plots and diagrams presented on the AceView server [6]. It provides among other tools a graphic interface helping to compose complex tables. But the description of the graphic acedb system is out of the scope of this user guide.

The MAGIC pipeline [7] is a good example of a complex system relying on acedb.

A.5 C language API

The acedb system has a C language API. The application code can either be linked into the open-source database server code, in which case it directly shares the caches of the database engine, or it can be part of a client code which calls the server via tcp. The interesting point is that exactly the same code works in both situations, one changes from a server to a client code by simply changing the library used at link stage. Here is an example of a complete C program invoking the query language

```
#include <ac.h>  /* acedb C language API header */

int main (int argc, const char **argv)
{
  AC_HANDLE h = ac_new_handle () ;
  const char *errors = 0 ;
  const char *dbNameS = "acedb_directory" ;              // server case
  const char *dbNameC = "t:machine_name:port_number" ;   // client case
  AC_DB db = ac_open_db (dbNameX, &errors);              // X = S|C
  const char *query = "select p in class person where p like a*" ;
  AC_KEYSET aks = 0 ;  /* initial keyset, populates the @ symbol */
  AC_TABLE t = ac_bql_table (db, query, aks, errors, h) ;

  printf ("Found % lines in the table\n", t->rows) ;

```

```
16    ac_db_close (db) ;
17    ac_free (h) ;
18    return 0 ;
19  } /* main */
```

The interface is detailled in the self documented header file wh/ac.h.

All comments are welcome.