# AceC, a C programmer's interface to the AceDB database system.

Mark Sienkiewicz and Jean Thierry-Mieg
NCBI
*The AceC code was first released in 2004*
*This document was last revised March 27, 2007*

## *Abstract*

AceC is a simple and fast programmer interface to the AceDB object oriented database system. It can be used to write a light client code which connects at the same time to one or more AceDB servers, to write stand alone applications directly reading an AceDB disk, or to write subroutines that may be imbedded inside the main AceDB library. In client mode, speed is obtained by implicitly running the data intensive task on the server. In embedded mode, AceC is just a wrapper around AceDB kernel calls, and therefore runs without any appreciable overhead.
AceC is implemented as a C language library, and is part of the distribution of the NCBI version of the AceDB freeware

## *Introduction*

AceDB is an object oriented database management system with many applications in bioinformatics [1-27]. Its main qualities are its data definition language, its rich graphic interface and its speed. These 3 characteristics are intertwined. Because it relies on a powerful grammar, reminiscent of O2, an AceDB schema is much shorter than the equivalent relational schema and is easier to understand. This implies that fancy applications are simpler to write and to maintain when the schema needs modification, and also that data handling and queries can be well optimized at the kernel level, because they are *a priori* less convoluted and involve fewer joins. Another asset is the .ace data exchange format. Objects can be validated, imported and exported in and out of AceDB, using simple text files, which contain no parenthesis or other cryptic signs and are therefore easy to read, write and manipulate using

line oriented tools like awk, grep and sed. For all these reasons, AceDB is popular among data curators.

The problem however is that it is difficult to develop customized add-on applications. AceDB is monolithic. The way to add a new functionality is to insert a new option somewhere inside the command line interface or to modify one of the graphic menus. This is both difficult to do and nearly impossible to maintain. The user must either repeat the modification with each new release of the AceDB kernel or convince the kernel developers at AceDB that this new functionality is of general interest, works on all platforms, and should be integrated in the main distribution.

A major improvement came in 1999, with the implementation of AcePerl in collaboration with Lincoln Stein [13]. AcePerl is an AceDB database client library for the Perl language. Perl is very popular in both bioinformatics and web based applications. AcePerl is used underneath most AceDB web sites. AcePerl takes advantage of the fact that Perl can execute code constructed on the fly to provide a seamless fusion between the embedding language, Perl, and the AceDB database being queried. For example, one may ask for db->Gene->Sequence->asDna(), where db and asDna() are objects and methods defined in AcePerl, whereas Gene and Sequence are defined only in the AceDB schema of the particular database server you are connected to. This is nice, but sometimes slow, and Perl is not always the most convenient language.

The new system presented here, AceC, is a C programmer's interface to AceDB. It is not a re-coding of AcePerl for the C language, but a complete re-implementation. AceC has a number of advantageous characteristics:

- The interface is intended to be complete and yet easy to document, understand, manipulate and maintain.

- The same code can run either as a database client (client mode), as part of a subroutine inside tace or xace (subroutine mode), or as a stand-alone application that directly reads the AceDB disks (stand-alone mode).

- Speed is optimized. In stand alone or subroutine mode, there is no measurable overhead. In client mode, the amount of data exchanged with the server is minimized.

AceC should be chosen whenever performances are crucial or if C is deemed preferable to Perl. Actually, we could accelerate AcePerl by using AceC underneath.

### *About the design of AceC*

The first idea is to provide the user with a clear cut interface, easy to document, understand, manipulate and maintain. The data is presented to the user as basic types: object, integer, float, date or text; as rectangular tables containing basic types; or as list of objects called keysets and iterators. There is no public distinction between an object identifier and an object reference. Furthermore, the tree structure of the AceDB objects is hidden from the user who sees branches of AceDB objects as small tables. Memory handling, a standard headache in C language, is made easy by providing a single call to release any kind of memory, and the possibility to attach all the memory created by AceC calls to so-called memory handles which can be freed as a whole. This minimal library was chosen by analyzing around fifty thousand lines of existing AceDB subroutines in order to be sure that our new library would provide exactly one way of obtaining the same results.

The second idea is to be able to run the exact same code either as a client, or linked within the AceDB kernel lib. For this purpose, we provide two implementations of the same interface, libacs.a and libaccl.a (short for LIBrary ACe Server and CLient). Linking with libaccl.a, you construct a C client able to talk to one or several AceDB servers. Linking with libacs.a, your new subroutines are compatible with the AceDB main program, xace or tace. In the latter case, AceC does not exchange data with the AceDB kernel, but directly shares the kernel caches. In fact the libacs.a library does no real work but simply wraps the very numerous AceDB kernel calls so that they appear to the user under the disguise of the simple AceC interface, with no appreciable overhead.

The third idea is to pay great attention to performances. In embedded mode, as we have just seen, AceC is as fast as the AceDB kernel. In client mode, the critical issues are the number and the size of the transactions. To minimize their numbers, we define 2 types of lists of objects: iterators and

keysets. When iterators are used to explore on the client side a possibly long list of objects, data are transferred to the client using blocked lazy loading. Keysets, on the other hand, provoke no network activity, but yet allow complex manipulations of the data. They are defined in intention on the client side and in extension only on the server side. For example, you can construct several queries and take the intersection of the replies without ever passing the actual lists of objects across the network.

To minimize the size of the transactions, we have defined a new data exchange format, called show -C, building upon discussions long ago with Doug Bigwood, Sam Cartinhour and Richard Durbin and implemented by Vahan Simonyan while he was working on a predecessor of AceC. Numbers and Tags (AceDB field identifiers) are transmitted in binary mode, name and texts as normal ASCII character strings and the geometry of the AceDB object by one-byte directional commands before each new data item. This is 30% shorter than the show -p used in AcePerl and easier to interpret on the client side. Byte swapping is performed automatically if needed.

Considering the problem of clients requiring write access, we chose a very simple approach. The AceC clients work in asynchronous read-only mode, then transmit data back to the server as .ace files. To resynchronize and see the modified data, a client has to explicitly refresh its local caches. This is naive but robust, even when the connection is unreliable, and offers little possibility of programming errors on the client side. Of course, in stand alone or subroutine mode, AceC sees the exact current state of the data and there is no synchronization issue.

## *Using AceC*

The objective of this paper is to discuss the main concepts underlying the design of AceC, so that present or potential users of AceDB can understand how they may use this library. We even hope that some of these ideas will be interesting in other contexts. For these reasons, we prefer not to overflow the reader with too many technicalities. The descriptions below are not exhaustive; we do not discuss all the available functions and all their parameters. For more details, please refer to the AceC documentation which is included in the wac directory of the NCBI AceDB distribution.

## Communication protocols and Makefiles

In client mode, the first parameter (a: in this example) indicates the transport protocol chosen by the server, which may be 'a', 's' or 'r'. The 'a' protocol is the most efficient (faster, less traffic) but it is only supported by AceC and the NCBI distribution of the AceDB server. The 'r' (rpc) protocol is known to the AceC and AcePerl, it is supported by the NCBI and by the Sanger distribution of the AceDB server. The 's' (Sanger) protocol is known to AceC and AcePerl, but only supported by the Sanger AceDB server. The protocols and the correlative configuration of the AceDB server are described in the file 'wac/README' of the NCBI AceDB distribution.

To compile an AceC program in standalone or client mode, the only thing to do is to change the library you link to. The simplified command lines

```
cc –o client_example          libacs.a  acec_example.c
cc –o stand_alone_example     libaccl.a acec_example.c
```

will compile a client (speaking the 3 protocols) and a stand-alone executable from the same source code. The actual compilation command must specify some machine specific details (linux, macX, 32 or 64 bits etc); many working examples are given in the directory 'wacext' of the AceDB NCBI distribution.

## *How to program in AceC*

AceC defines 6 types of pointers AC_DB, AC_HANDLE, AC_OBJ, AC_TABLE, AC_KEYSET and AC_ITER and their associated methods described below.
All function names are composed in the same way. They start with ac, to characterize the library. The second part of the name characterizes the action or the main parameter of the function and the third part the type of the result. If the function needs to allocate memory, its last parameter is a AC_HANDLE. For example ac_keyset_table (ks, h) transforms a keyset ks into a one-column table, allocated on the handle h.

### AC_DB: Opening and closing the connection.

The first thing you have to do when using AceC is to open a connection by calling

AC_DB ac_open_db (char *target, char **error);

If the connection fails, ac_open_db returns NULL and the reason why it failed is documented in the optional error field. If the connection succeeds, ac_open_db returns a handle of type AC_DB which must be passed in subsequent calls to AceC, and finally to ac_db_close in order to close the database connection.

Example:

```
{
 char *error = 0, target =
    "a:localhost:12345:anonymous" ;
 AC_DB db = ac_open_db (target, &error) ;

 if (db) {
    getOranges (db) ; /* shown below */
    ac_db_close (db) ;
 } else {
   printf ("cannot connect to %s, error %s\n",
    target, error) ;
  }
}
```

In client mode, the target should be
a:hostname:port[:username/password]
where hostname can be blank to indicate 'localhost' and the port, the user name and the optional password must correspond to the way the AceDB server is configured.
In standalone mode, target should be the path to the AceDB database directory, i.e. the same $ACEDB parameter which is normally provided to the tace and xace commands.
If you write a subroutine to be linked within the AceDB program, you must still call ac_open_db, however the call cannot fail, the 2 parameters are ignored, and you see the same database as the entire program.

## Writing data in the database

The only write operation in AceC is:
-ac_parse
It takes as arguments a db handle and a text, formatted as an ace file, an error handle, a keyset and a memory handle. It returns true if parsing is successful and lists all parsed objects in the keyset. It returns false and an error message otherwise. To be able to parse some data, you must have write access on the server. How to do so is explained in the server configuration documentation. Each call to ac_parse is independent, and must contain a valid set of .ace paragraphs. After editing data on the server side, to see the modified data on the client side, you must call
ac_db_refresh
This ensure that the next time you open an object on the client side, you will get a fresh copy from the server, not a copy recovered from the client cache. However, remember that the objects which are already open on the client side are not modified by the call to ac_parse.

Example:

```
static void birthDay (AC_DB db, AC_OBJ person)
{
/* assume the schema
  ?Person Age UNIQUE Int
 */
  AC_HANDLE h = ac_new_handle () ;
   int age = ac_tag_int (person, "Age", 0) ; /* 0 is the default value */
   char *buff[1000], *error_message = 0 ;
  AC_KEYSET ks = ac_new_keyset (db, h) ;

   sprintf (buff, "Person \"%s\"\nAge %d\n\n", ac_name(person), ++age) ;
  if (ac_parse (db, buff, error_message, ks, h))
{
    ac_db_refresh (db) ; /* so that new_person will differ from person */

    new_person = ac_get_obj (db, ac_class (person), ac_name(person), h) ;
    printf ("old age %d\n", ac_tag_int (person, "Age", 0) ;
    printf ("new age %d\n",  ac_tag_int (new_person, "Age", 0) ;
```

```
}
else
  printf ("Sorry, ac_parse failed error=%s\n", error_message) ;

ac_free (h) ;
}
```

## AC_HANDLE: memory allocation

Memory leaks and garbage collection is a standard headache. In some programming language, like Perl, releasing memory is completely automatic. This is nice for the programmer, but often expensive in terms of execution time and memory requirements, because the actual release may happen late or never. In C, every operation must be programmed explicitly. The resulting code is very fast, but often contains memory leaks. AceC offers an intermediate solution.

Any time you create an AC_* pointer, you must free it by calling ac_free. When you do so, it immediately becomes invalid and all memory dependent on that object also dies. However, you can very much simplify the bookkeeping by creating an AC_HANDLE, using ac_new_handle(), and passing it as a parameter to all the functions which may allocate memory. When you free the handle, using ac_free(), all memory allocated on the handle is released at the same time. See the examples below.

## AC_OBJ: Object manipulation

Data in AceDB is organized in objects, following a rich schema. The advantage is that it is relatively easy to represent in AceDB the complex data needed for example in biology. The drawback is that the programmers interface, to access inside the so called constructed types, is complex, both in the AceDB kernel, and in AcePerl. There are several ways to do the same thing and it is not always immediately clear why this or that method works or does not work. In AceC, we adopted a simplified interface. First, we provide direct calls which follow the tag-value paradigm. They implement the very frequent operations of assessing the existence of a tag and of retrieving the first data element associated to this tag, which is sufficient if the tag is single valued, UNIQUE in AceDB terminology. Then, for anything more complicated, we treat the data as tables, described below.

There is a direct object constructor
- ac_get_obj: which requests from the server a single object with known class and name

and 2 naming functions:
ac_name: gives the name of the object
ac_class: gives the name of the parent class of the object.
The returned values point inside the object and go away when the object is freed.

But most of the time, objects are constructed from another AceC structure:
- ac_copy_obj: to duplicate (the reference to) an existing object
- ac_iter_obj: discussed below, when looping on an iterator,
- ac_tag_obj: to navigate from one object to its neighbor,
- ac_table_obj: discussed below, to access an object mentioned in a table.

To access the tag-value pairs, you can use
- ac_has_tag: to assess the existence of a tag
- ac_tag_type: to know the type of data following the tag
- ac_tag_*, with * a fundamental type: obj, int, float, date, key, text
- ac_tag_printable: to get the data item as a volatile printable string.

Example:

```
void displayOneOrange (AC_OBJ obj)
{
  /* violet is the default, it will be used if obj has no color */
  printf ("The first color in object class=%s, name=%s is %s\n",
            ac_class (obj),
            ac_name (obj),
            ac_tag_printable (obj, "Colour", "violet") ) ;
 if ( ac_has_tag (obj, "Colour"))
    printf("This object has a colour attribute\n") ;
}
```

## AC_TABLE: Tables

Tables are fundamental in database design because they provide an excellent way to store and retrieve simple data structures. Although we have designed AceDB in an object oriented way, because we believe that object schema are more versatile and easier to maintain when they become very large, as needed for example in genome applications, we are aware that, when one is writing a particular client function, one usually wishes to handle a particular type of data, and for that purpose, tables are optimal. Therefore, we have introduced in AceC the AC_TABLE structures and use it in multiple ways.

There are several constructors:
ac_empty_table: returns an empty table, optionally allocated on a handle.

ac_obj_table: returns a complete object as a table. Column 0 will contain the level 1 tags of the objects, column 1, the level 2 tags and some data, and so on. Since the schema may be complex, one expects most cells on the right to be empty.

ac_tag_table: returns as a table the branch of the given object, to the right of the given tag. This is a very frequent operation.

ac_keyset_table: creates a one column table, in alpha-numeric order. The main difference between a keyset and a table is that the keyset exists on the server side but the table exists on the client side. It is possible to get the name of the third object in the table using
ac_name(ac_table_obj(table,0,2,h)); /* column 0, line 2, allocated on handle h */ whereas there is no direct way to access a member of a keyset.
By the way, we see here a nice usage of the memory handle h. The function ac_table_obj allocates an AC_OBJ. Without h, we should either bookmark the retuned object and free it explicitly or suffer a memory leak. Using h, we can perform a number of such implicit creations and free them all at the same time.

ac_aql_table: returns a table constructed on the server side using the AQL query language, which is nice and clean if you are analyzing a small database, and if you know how to write an AQL query.

ac_tablemaker_table: returns a table constructed on the server side using the AceDB table-maker interface. The drawback is that a table-maker query must be defined in advance and stored either on disk or as an AceDB object

and invoked by name. But vice versa, the advantages are that AceDB does have a nice graphic interface to define the table, and that the table-maker search is believed to be bug free and is much better optimized than AQL. Table-maker is strongly recommended if speed and memory are an issue.

The geometry of the recovered table is given by ac_table_cols/ac_table_rows: they return 1/1 if the table contains a single cell. The table column/row indices run, as traditional in C, from 0 to ac_table_cols/rows() - 1.

To access the data present in the table, we provide the functions ac_table_* with * one of:
- type: to know the type of data in that particular cell. Remember that table columns are not always homogeneous, for example if obtained from ac_obj_table()
- tag, obj, int, float, date, key, text: to see the data.
- printable: to get the data item as a printable string (its name in the case of an object)

Example:

```
void displayAnotherOrange (AC_OBJ obj) /* version 2 */
 {
AC_TABLE tbl = ac_tag_table (obj, "Colour", 0) ;
int i ;

for (i=0; i < tbl->rows; i++)
   printf ("The %d the color in object %s:%s is %s\n",
              i,
              ac_class (obj),
              ac_name (obj),
              ac_table_printable (tbl, i, 0));
 ac_free (tbl) ;
}
```

## AC_KEYSET: Sets of objects defined on the server side

Keysets are fundamental tools of the AceDB system. As the name tries to indicate, they are sets of object identifiers, i.e. non repetitive collections. In

graphic AceDB, xace, they are constructed each time you ask a query and shown in ubiquitous 'keyset windows', which you may intersect, unionize, save, restore etc., and then use as the starting point of further operations like new queries, dumps and tables. We gave them a fundamental role in AceC, and optimized their usage, by keeping them completely on the server side.

a) There are 7 constructors called ac_*_keyset where the star can be:
- new: creates a new empty keyset,
- copy: copies an existing keyset,
- dbquery: runs a query against the whole database, the most frequent operation,
- objquery runs a query initialized on a single object
- ksquery runs a query initialized on a previous keyset
- command: gets the active keyset resulting from any AceDB command,
- read: reads a server accessible disk file, keep the recognized objects.

b) We then provide a count utility, a subset constructor, the 4 standard logical operators, a disk write operation and 2 modifiers. The functions are called ac_keyset_* where the star can be
- count: gives the number of elements in the keyset,
- subset: gives a slice of the keyset in AceDB alphanumeric order,
- and, or, xor, minus: modify their first parameter, and return its new number of elements, rather than creating a new keyset, because we found in the review of existing code that this was the most frequently desired behavior, for example when computing multiple intersects. minus means suppress from the first keyset all objects contained in the second one.
- read: reads a keyset from disk, previously written by ac_keyset_write()
- add, remove: take a single object as second parameter and add/remove it to/from the set.

Example

```
	void getOranges (AC_DB db)
{
int nOranges ;
AC_HANDLE h = ac_new_handle () ;
AC_KEYSET reds, yellows, oranges ;
		/* assuming colors are defined in the schema */
```

```
reds = ac_query_keyset (" Find Fruit Colour = Red", h) ;
yellows = ac_query_keyset ("Find Fruit Colour = Yellow", h) ;
oranges = ac_keyset_copy (reds, h) ;                    /* copy first */
nOranges = ac_keyset_and (oranges, yellows) ;        /* modify oranges */

printf ("%d fruits are red and yellow, they may be oranges\n", nOranges) ;
displayOranges (oranges) ;                              /* defined below */

ac_free (h) ;       /* releases all the memory allocated in this subroutine */
}
```

## AC_ITER: Iterators sets of objects defined on the client side

Using iterators is a polite way to scan large sets of objects. They can be constructed from a keyset, or, for convenience, directly from a query. Then you loop on ac_next_obj until that function returns a NULL. ac_iter_rewind can be used to reinitialize loops on the same iterator. Objects are imported from the server in blocks, which is more efficient than importing them one by one, but not all at the same time, which could swamp the client memory and would return only at the end of a possibly very long server transaction.

Example:

```
void displayOranges (AC_KEYSET ks)
{
 AC_ITER iter = ac_keyset_iter (ks, 0) ;
AC_OBJ obj ;

while ((obj = ac_next_obj (iter)))  { /* loops through */
   displayOneOrange (obj) ; /* defined above */
   displayAnotheOrange (obj) ; /* defined above */
   ac_free (obj) ;
}
ac_free (iter) ;
}
```

## DNA, protein sequences and long texts

Apart from the tree objects described in the schema, as described above, AceDB supports several type of so called A classes. The most important are the LongText class, used to store paper abstracts, and the DNA and peptide classes, uses to store biological sequences. Any A class can be seen in AceC using the ac_a_data, but for the 3 classes just listed, we provide a direct interface:
- ac_longtext: recovers, as a single char* the content of a longtext,
- ac_dna: recovers as a char* the DNA sequence associated to an object
- ac_zone_dna: allows to recover a particular subsequence, for example a genomic sequence upstream of a given gene. If the coordinates are given backwards, the sequence is automatically reverse-complemented.
- ac_pep: gives the translation of a gene, or directly the sequence of a protein object

Example:

```
static void showAbstract (AC_OBJ paper)
{
  AC_OBJ abs = ac_tag_obj (paper, "Abstract", 0) ; /* recover the abstract
identifier */
  char *abstract = 0 ;

 if (abs) { abstract =  ac_longtext (abs, 0) ;
            printf ("%s\n", abstract) ;
            ac_free (abstract) ; ac_free (abs) ;}
}

static void showPromotorRegion (AC_OBJ gene)
{ /* we assume the simple schema
     ?Gene Parent ?Chromosome XREF Gene
     ?Chromosome Gene ?Gene XREF Parent Int Int
    where the 2 integers denote the coordinates of the gene on the
chromosome.
 */
  AC_HANDLE h = ac_new_handle () ;
  AC_OBJ parent = ac_tag_obj (gene, "Parent", h) ;
  AC_TABLE genes = ac_tag_table (parent, "Gene", h) ;
  int ir, x1, x2 ;
 char *dna ;
```

```
  for (ir = 0 ; ir < genes->rows ; ir++)
   {
     if (strcmp (ac_name(gene), ac_table_printable (genes, ir, 0)))
        continue ; /* locate the required gene */
      x1 = ac_table_int (subseq, ir, 1, 0) ; x2 = ac_table_int (subseq, ir, 2, 0);
      if (x1 < x2) /* gene on down strand */
          dna = ac_zone_dna (parent, x1 - 1000, x1-1, h) ;
     else /* gene on reverse strand */
          dna = ac_zone_dna (parent, x1 + 1000, x1+1, h) ;
     printf ("1000 base upstream = %s\n", dna) ;
   }
 ac_free (h) ;
}
```

**Other functions:**

Because they are needed in its own implementation, AceC automatically links the AceDB tool libraries providing numerous utilities, all highly optimized and thoroughly tested. They include, for example, the memory handle package, the array package, the dict and the AceDB i/o package. One advantage to use these, rather than their glib equivalent, is to minimize the overall size of the client program, which will therefore load faster, a crucial factor if you are writing a web cgi script.

## *Conclusion*

AceC provides a minimal set of functions providing a standardized way to access AceDB databases. Their names are easy to remember, they are called ac_xxx_yyy with xxx the characterizing the principal parameter and yyy recalling the type of the result. All the functions allocating memory take a memory handle as last parameter, which allows global memory management. At the same time, we have tried to minimize the number of parameters in each function call and to remove all duplicate ways of doing the same thing. All of these are well known recipes, and yes they work, the resulting client code is easy to understand and to maintain.

Speed was, after clarity, our second most important target. So far, we have limited experience, but we did accelerate our web site AceView a comprehensive annotation of human and worm genes with mRNAs or ESTs by at least a factor 4 by switching from AcePerl to AceC.
Finally, in imbedded mode, all our recent AceDB development uses the AceC library, with no appreciable speed overhead, but with the advantage of a standardized interface which makes the code easier to write and to maintain. We could verify in this way that the library offers all the needed functionalities.

**Acknowledgements:**

We would first like to thank Vahan Simonyan who wrote a precursor of AceC, which helped us to define our needs. We are also grateful to the whole AceDB team, which over the years has helped us to ameliorate and optimize the AceDB kernel, in particular Simon Kelley who conceived the memory handling package used everywhere in AceC, Lincoln Stein for the development of the Java and Perl interfaces to AceDB from which we borrowed many ideas, Michel Potdevin for innumerable discussions, Richard Durbin for the fundamental design of AceDB and Danielle Thierry-Mieg whose everlasting demand for a better biological analysis of the data has motivated the development of a faster and more versatile interface to AceDB.

Bibliography

1: Lebeda FJ.
BotDB: A database resource for the clostridial neurotoxins.
Mov Disord. 2004 Mar;19 Suppl 8:S35-41.

2: Srinivasan J, Otto GW, Kahlow U, Geisler R, Sommer RJ.
AppaDB: an AcedB database for the nematode satellite organism Pristionchus pacificus.
Nucleic Acids Res. 2004 Jan 1;32 Database issue:D421-2.

3: Leveugle M, Prat K, Perrier N, Birnbaum D, Coulier F.
ParaDB: a tool for paralogy mapping in vertebrate genomes.
Nucleic Acids Res. 2003 Jan 1;31(1):63-7.

4: Le Hellard S, Ballereau SJ, Visscher PM, Torrance HS, Pinson J, Morris SW, Thomson ML, Semple CA, Muir WJ, Blackwood DH, Porteous DJ, Evans KL.
SNP genotyping on pooled DNAs: comparison of genotyping technologies and a semi automated method for data storage and analysis.
Nucleic Acids Res. 2002 Aug 1;30(15):e74.

5: Martin SL, Blackmon BP, Rajagopalan R, Houfek TD, Sceeles RG, Denn SO, Mitchell TK, Brown DE, Wing RA, Dean RA.
MagnaportheDB: a federated solution for integrating physical and genetic map data with BAC end derived sequences for the rice blast fungus Magnaporthe grisea.
Nucleic Acids Res. 2002 Jan 1;30(1):121-4.

6: Stein L, Sternberg P, Durbin R, Thierry-Mieg J, Spieth J.
WormBase: network access to the genome and biology of Caenorhabditis elegans.
Nucleic Acids Res. 2001 Jan 1;29(1):82-6.

7: Quentin Y, Fichant G.
ABCdb: an ABC transporter database.
J Mol Microbiol Biotechnol. 2000 Oct;2(4):501-4.

8: Wixon J.
Website review: UK CropNet.
Yeast. 2000 Sep 30;17(3):244-54.

9: Kelley S.
Getting started with AceDB.
Brief Bioinform. 2000 May;1(2):131-7.

10: Pollet N, Schmidt HA, Gawantka V, Vingron M, Niehrs C.
Axeldb: a Xenopus laevis database focusing on gene expression.
Nucleic Acids Res. 2000 Jan 1;28(1):139-40.

11: Nakata K, Takai T, Kaminuma T.
Development of the receptor database (RDB): application to the endocrine disruptor problem.

Bioinformatics. 1999 Jul-Aug;15(7-8):544-52.

12: Blackwell JM, Melville SE.
Status of protozoan genome analysis: trypanosomatids.
Parasitology. 1999;118 Suppl:S11-4. Review.

13: Stein LD, Thierry-Mieg J.
Scriptable access to the Caenorhabditis elegans genome sequence and other ACEDB databases.
Genome Res. 1998 Dec;8(12):1308-15.

15: Stein LD, Cartinhour S, Thierry-Mieg D, Thierry-Mieg J.
JADE: An approach for interconnecting bioinformatics databases
Gene. 1998 Mar 16;209(1-2):39-43.

16: Takai-Igarashi T, Nadaoka Y, Kaminuma T.
A database for cell signaling networks.
J Comput Biol. 1998 Winter;5(4):747-54.

17: Walsh S, Anderson M, Cartinhour SW.
ACEDB: a database for genome information.
Methods Biochem Anal. 1998;39:299-318. No abstract available.

18: Cousin X, Hotelier T, Giles K, Toutant JP, Chatonnet A.
aCHEdb: the database system for ESTHER, the alpha/beta fold family of proteins and the Cholinesterase gene server.
Nucleic Acids Res. 1998 Jan 1;26(1):226-8.

19: Degrave W, de Miranda AB, Amorim A, Brandao A, Aslett M, Vandeyar M.
TcruziDB, an integrated database, and the WWW information server for the Trypanosoma cruzi genome project.
Mem Inst Oswaldo Cruz. 1997 Nov-Dec;92(6):805-9.

20: Miller G, Fuchs R, Lai E.
IMAGE cDNA clones, UniGene clustering, and ACeDB: an integrated resource for expressed sequence information.
Genome Res. 1997 Oct;7(10):1027-32.

21: Martinelli SD, Brown CG, Durbin R.
Gene expression and development databases for C. elegans
Semin Cell Dev Biol. 1997 Oct;8(5):459-67.

22: Povey S, Attwood J, Chadwick B, Frezal J, Haines JL, Knowles M, Kwiatkowski DJ, Olopade OI, Slaugenhaupt S, Spurr NK, Smith M, Steel K, White JA, Pericak-Vance MA.
Report on the Fifth International Workshop on Chromosome 9 held at Eynsham, Oxfordshire, UK, September 4-6, 1996.
Ann Hum Genet. 1997 May;61 ( Pt 3):183-206.

23: Igarashi T, Kaminuma T.
Development of a cell signaling networks database.
Pac Symp Biocomput. 1997;:187-97.

24: Dunham I, Maslen GL.
Use of ACEDB as a database for YAC library data management.
Methods Mol Biol. 1996;54:253-80. No abstract available.

25: Waterston R, Sulston J.
The genome of Caenorhabditis elegans.
Proc Natl Acad Sci U S A. 1995 Nov 21;92(24):10836-40.

26: Eeckman FH, Durbin R.
ACeDB and macace.
Methods Cell Biol. 1995;48:583-605. Review. No abstract available.

27: Bergh S, Cole ST.
MycDB: an integrated mycobacterial database.
Mol Microbiol. 1994 May;12(4):517-34.