

Turbo Charge your 4D Database

Presented by: **Thibaud Arguillère & Olivier Deschannels**



Knowing how 4D stores data, organizes the data file (« .4DD » file) and uses indexes is important for two main reasons:

- Understanding how our development tool works is part of the Fundamental Knowledge,
- It will help to improve the performances (speed and/or storage volume) of our database, and it sometimes can dramatically improve it.

As it is a huge subject, we divided it in two sessions: the first one will focus on explanations and fundamental knowledge about records storage and indexes; the second session will focus on optimizations (using indexes in queries and sorts, choosing types of fields, ...). In the current document, we'll focus on the main explanations about data storage, the sessions go more in-depth. A component is provided, to illustrate the concepts and to help you to analyze your data.

A BRIEF REMINDER ABOUT THE DATA FILE IN 4D

The history of the internal structure of data/index files has three steps:

- One data file plus one index file per index (the good old French v3)
- One « .data » file, holding the records and the indexes. Even if, internally, things did change between major versions, even if we could add segments to the data, this way of organizing data has been used up to 4D 2004 (included).
- One single data file and one single index file: this is the current architecture in 4D v11 SQL. Of course, it may change (and it probably will).

THE .4DD FILE IN 4D V11 SQL: A BLOCK HISTORY

The .4DD file (and, as we'll discuss later, the .4Dindx file) is organized in blocks. All of the data file is nothing more, nothing less than a succession of blocks. All the blocks have the exact same size: 128 bytes. So, for example, if you « Get info/properties for » a data file from the desktop and you divide the size by 128, you'll get the exact amount of blocks this data file is using (at this level of the session, this is not that useful).

Different kinds of information can be stored in blocks. Most of the blocks hold records, of course. But an important (logically speaking) part of the blocks store information about records and not the records themselves. More precisely: information about the locations of the records in the file.

There are four main things to know about those blocks:

1. 128 is the minimum: Even if the logical size of an information is less than 128 bytes (say a record of a table with a single short integer field), it will use 128 bytes. This 128-byte limit can't be changed by the developer.
2. One block stores only one kind of information: A block can't mix information: it stores a record or an index or a bitmap or ...
3. Information uses contiguous blocks: If the information to store takes more than 128 bytes, then a second block must be used and it must be contiguous to the first one. Say a record has a weight of 1 280 bytes (including the internal and private information used by 4D that we'll explain later), 4D will store it in 10 contiguous blocks.
4. All blocks share the same kind of internal structure: They start with a header, followed by the data itself. The header starts with a tag that gives the kind of the stored information (ie « rec1 » for a record). The header exists only in the first block when the information uses more than one block (it is not written again on each contiguous block)

STORAGE OF RECORDS: HEADER, DATA, AND MICROSTRUCTURE

1) The header

The header of a block that stores a record has a fixed size of 32 bytes (that changed from 4D 2004). It starts with the usual *block tag* (« rec1 » in this case). The tag is followed by miscellaneous values:

- About the table it belongs to,
- About the size of the data (the exact, logical size),
- And some other, but important, information: a checksum of the data and a timestamp

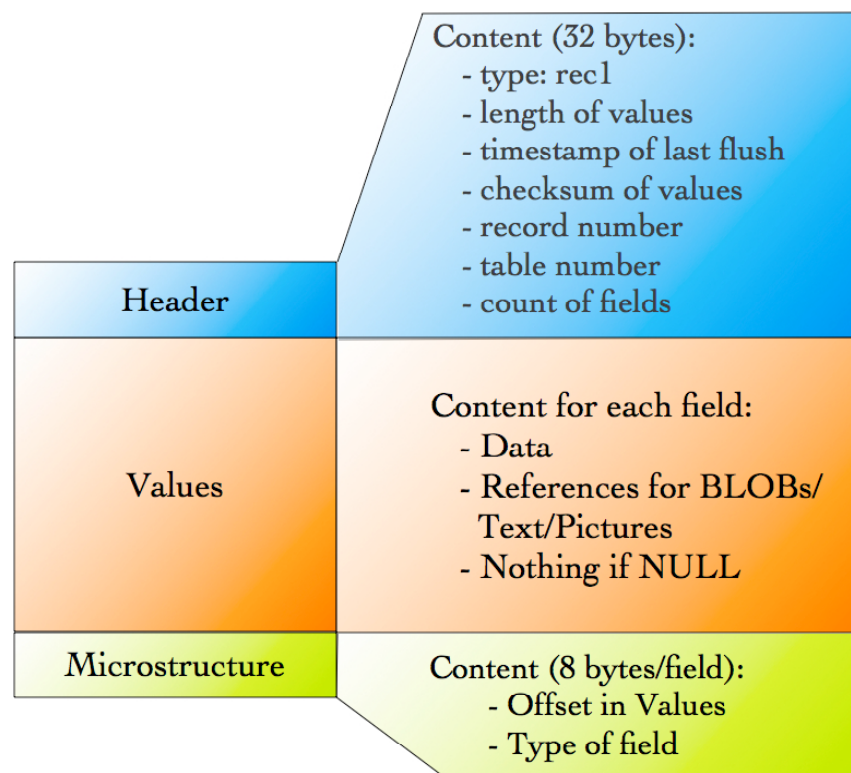
All those information let 4D to know exactly what it's going to load (or save), and it permit to check the data before loading it, to detect a problem when the checksum looks invalid while loading the record. The timestamp concerns flushing the record from the cache to the disk, it is not the date/time when the developer called SAVE RECORD, or a new record is added using the import dialog, etc.

2) The values

All values of fields are stored one after the other, in the order of the creation of the fields. When a variable sized field (Text/Blob/Picture) is stored outside the record (see the field properties), then it is a reference to this external storage that is stored.

3) The microstructure

At the end of the block (or the blocks if there are several), we find what is called a *microstructure*. This microstructure has a size of 8 bytes per field (this is an other change from 4D 2004), and for each field it maintains two main information: the type of the field and the offset of the field's data in the *values* part of the block. It also holds some internal informations (it uses some bytes of the type bytes for this purpose) that helps to optimize the storage (for example: « is this field null? »)



More about the microstructure

This part of the record's block explain some parts of 4D's power and why it is that easy-to-use. The microstructure saves the types of the field at the time the record is saved. When the record is

loaded, 4D also loads this microstructure and compares it to the actual table definition. If the kind of a field has been modified by the developer in design mode, then 4D automatically handles the appropriate conversion: say a field was a Date and it is now a long => 4D puts 0 in the field because it can't convert from a date to long. On the other hand, if you change the type from long to text, then the field will hold "123" instead of 123. Basically, 4D applies the appropriate conversion routine, just like if you wrote something like Date(theStringValue) of String(theNumValue).

Note that this conversion is done when loading the record, but it is not automatically saved: you must store the record explicitly (SAVE RECORD, an automatic navigation button, ...).

Of course, you may want to actualize the data of the table whose field was modified, because it's a good idea to reset the values to their original type, so queries, sort and loading are much faster. It's easy, you just need to...

ALL RECORDS([TheTable])

APPLY TO SELECTION([TheTable];[TheTable]aField:=[TheTable]aField)

...(with a table in read write mode, of course). And you don't need to use the modified field, you can use any other field of the table.

STORAGE OF RECORDS: EVALUATE THE SIZE ON DISK

Evaluating the size that a record will use on disk is the same as evaluating how many blocks it will use. At a first sight, one could state that the formula may look something like:

32 bytes for the header
 + Size of the microstructure (8 x count of fields)
 + Size of the values

 = Total logical size

Then, we just need to divide this by 128 to evaluate the count of blocks needed to store the record and, at the end, we know how many bytes our record really use in the data file (note: we don't take care of the size of the indexes here, they are not store in the same file).

In order to calculate this, we must then know what size take each field. For scalar values, it may look simple. « May look' because it is not that easy since alpha fields are considered as scalar while they are not. Here, « scalar » means precisely « not a Text, not a BLOB, and not a Picture field ».

For fields stored the record, the sizes are (including SQL types):

Type	Byte(s)
Boolean	1
Byte	1
Integer	2
Long	4

Long64	8
Real	8
Float	7 + n
Date	8
Time	8
Alpha	4 + (count of chars x 2)

Note: alpha fields are always stored in Unicode (UTF16), that's why it uses 2 bytes/chart

For large variable-sized fields, things are different, depending on the « Max. internal storage size » of the field property. If this property is 0, which means that the field must not be stored inside the record, then sizes are 4 bytes. Basically, those 4 bytes are the reference to the block that stores the real data.

Wherever they are stored, the data uses this count of bytes:

Type	Byte(s)
BLOB	4 + size of BLOB
Picture	4 + size of picture
Text	4 + (count of chars x 2)

Note: text fields are always stored in Unicode (UTF16), that's why it uses 2 bytes/chart

So, say a picture field has a switch size property of 3000:

- A record is stored, with a 2ko picture => it is stored inside the record's blocks and takes 2 052 bytes (4 + 2 048).
- Another record stores a 100 Ko pictures => it takes 4 bytes in the record's blocks, plus 102 432 bytes in the separate blocks (32 for the header + 4 + (100x1024))

ADDRESS TABLES

Now that we know how records are stored inside the data file, one question remains: how 4D retrieves those records when needed (query, load)? Simple: it uses *address tables*. Basically, the address table of a table can be seen as a map, that stores the location of each record of a given table inside the data file. Of course, the internals are more complicated, but this is the spirit. Note that we are not talking about indexes. The use of indexes to find a record comes *before* accessing the record.

So, address tables are stored in the data file, in contiguous blocks of 128 bytes, and they store:

- The physical address – inside the data file – of the records, using the record number as an index.
- The length of the data to read

We can see address table as a 2 columns array, one columns containing the address of every record of a given table, the second containing the size of the record's data. But, of course, this is not the way address tables are structured, because now that 4D can store one billion records per table, and even if a table contains « only » some dozen of millions of records, storing all the addresses/sizes in a single unit would lead to memory problems.

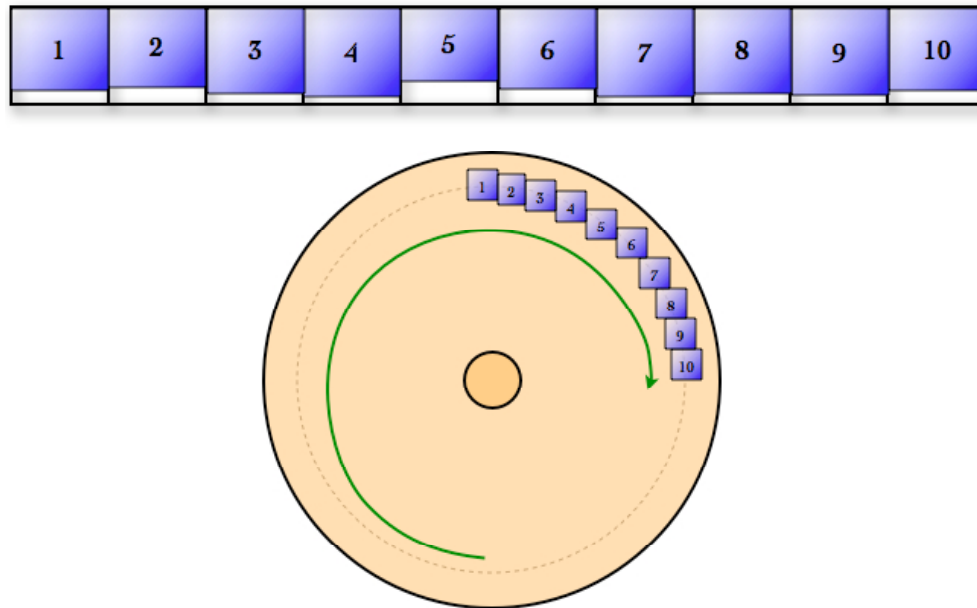
That is why address tables are themselves divided in primary/root tables and secondary address tables. At the end of this chain, we have the addresses of the records for a given page.

This mechanism is optimized: for example, if a table contains only 500 records, the table has a single address table because there is no need for a Root/Primary/Secondary address tables in this case.

Below, we'll use the term of Address Table without using the plural form.

FRAGMENTATION

Records are read from the address table. This is the sequential reading: 4D scans the address table and loads the records in this sequential, ordered way. As long as records are created, stored and never modified/deleted, this is fine because the hard disk head will follow the file naturally:



But data rarely evolves like that. More precisely, tables, inside a data file, are often modified. In the simple example of the previous picture, imagine a user loads record #2 and modify it in such a manner that it now needs 2 blocks. When 4D stores the record, it detects the record can't fit in its good old place and it then move it elsewhere. Say in two blocks just after record #10. Now, when 4D needs to sequentially read the records, the hard disk head: reads record #1, then jump to record #2, then goes back to record #3, etc.: the data is fragmented, accessing records is slow. One will not notice anything as long as a little part of the data is fragmented. And if the memory's cache is big, then this fragmentation may even never be noticed. But now, you know how this can happen.

The only way to un-fragment a data file is to compact it. This is easily done in the Maintenance Security Center, and can also be done « manually» (the good old: export all, delete indexes, import all, re-index).

INDEXES

Index are stored in the .4Dindx file. The structure of this file is exactly the same as the .4DD file: blocks of 128 bytes. But here, we store only indexes: values themselves (value + number of the corresponding record), and the needed tables handling the addresses of the different indexes tables.

We'll focus a bit more on the different kind of indexes.

1) The venerable BTree

We call it « venerable » because it is the way 4D handles indexes up to 4D 2004. Btree means « Balanced tree ». Basically, this kind of index stores values in *pages* of n values. When n is reached:

- The page is split in two sub pages, each one containing half of the values

- A root page is created, it holds the median value of the 2 pages and the address of those pages: values less than this median will be search in first sub page, values greater than this median value will be searched in the second sub page

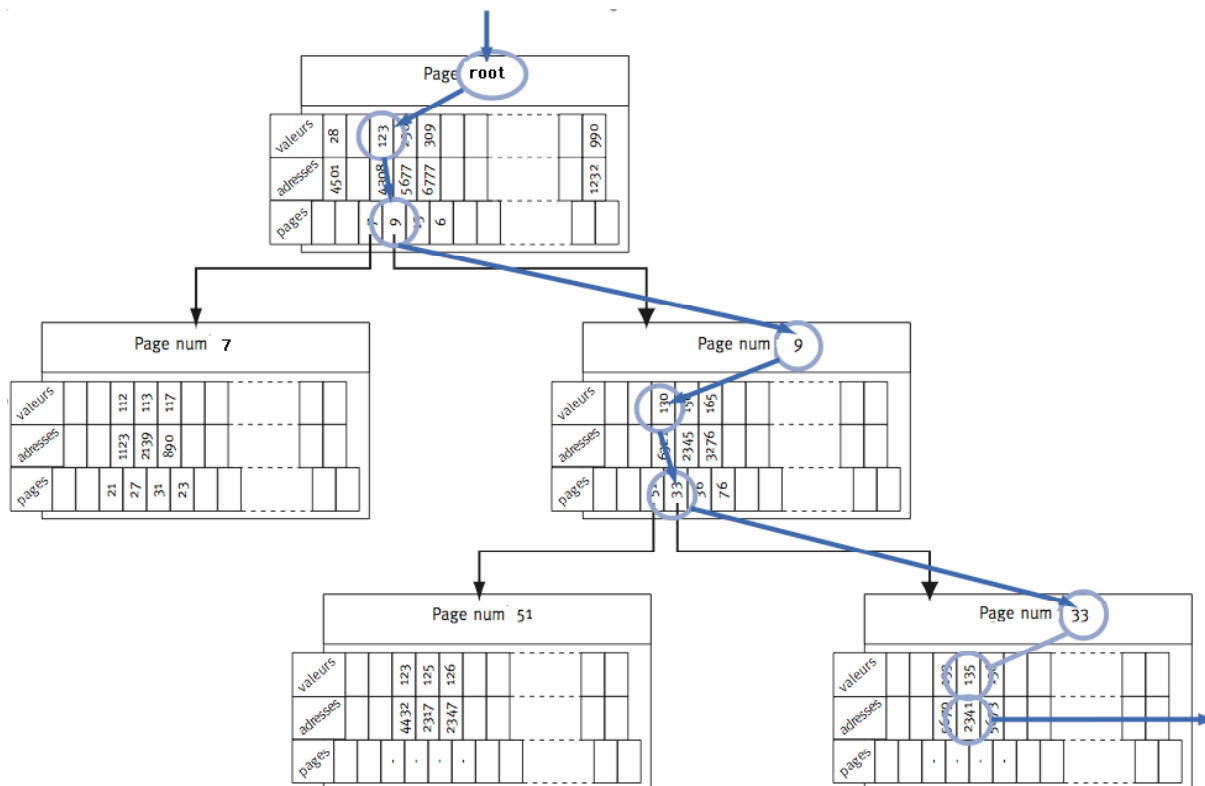
a *sub page* is created and the tree is (or may be) rebalanced, because (it's the definition of a BTree), it must stay ordered and balanced.

« Balanced » means that each branch/sub branch of the tree contains about the same count of values. And this is why when you modify 2, 3, 10 records it's fast and the day after it looks slow: 4D is probably re-balancing the tree.

Re-balancing the tree is not the same as rebuilding the index: values are not reloaded from the records.

Query inside a BTree looks like the following. Say we're searching for value 135 in a long (indexed) field;

- Value is not found in the root page. But we have 123 and 230, and the pages states that values between those 2 numbers are in index page 9
- So, we load index page 9. Value is still not here. We have 130 and 156. Between those 2 numbers, there is index page 33.
- So, we load index page 33, and find the value



2) The Cluster index

A cluster index stores things differently: for one index values, the cluster stores 1-n references to the records that match this values. There are basically 2 kinds of clusters: bit-table and selections.

The cluster bit-table

It stores infos about records matching the values just as a 4D set does: for each record macing the value, the bit corresponding to its number is set to 1, else it stays at 0. So, basically, for each index entry the size of the cluster is (count of records in the table / 8).

The cluster selection

This type of cluster stores an array of record numbers: its size is 4 x count of records matching the value.

Once those two kinds of clusters have been defined, it certainly sounds familiar to a 4D developer: we have here the same mechanism as the one found for Sets and Named selections in 4D.

That said, the developer can't set himself the kind of cluster to use, it is 4D which decides, because it optimizes storage as much as possible, using internal algorithms that we can resume as:

- It chooses the kind that uses the less space. For example if that table contains one million records and 10,000 matches a value, it will use a *selection*. Because $10,000 \times 4 \approx 40\text{Ko}$, while $1,000,000 / 8 \approx 125\text{Ko}$. On the other hand, if 500,000 records match a value, using the bittable will use less space.
- The previous algorithm is not a Rule because sets can be compressed. As an example, if a lot of contiguous values are the same (all 0s or all 1s) then 4D may still use a bittable.

3) How to choose?

Choosing between a BTree and a cluster may be sometime difficult, sometime obvious. If you need a frequent and quick access to a field that can take only 3-4 different values (« Mister », « Madam », « Miss »), then a cluster is the good choice. A boolean field may also be a good example. At the opposite, a unique-ID field should be a BTree, because all values are different, there is no reason to store clusters.

The choice can be valued by using an « Index Selectivity », which is the count of distinct values / count of records in the table. The smallest it is, the better a BTree is (a unique ID field as an index selectivity of 1).

4) The composite index

It is a concatenation of the values of 2 or more fields of the same table. The access to this index is not bijective, which means that, for example, when indexing Name + Surname, a query on those 2 fields or an order by (Name;Surname) will use the index while sorting by Surname + Name (Surname first) will not use it.