Terms

unlimited means somewhere in the order of $2^{32} - 1$ or $2^{31} - 1$

Index numbers, characters and addresses in graps increase to the right

0	1	2	3
foo	**bar**	**baz**	**Foobar**

→ increasing

Bits in values are ordered as MSB…LSB

Arrays

Arrays store data in one simple concecutive array of elements with no element gaps or pointers.

If an element is deleted, all successive elements move backwards to fill the gap; if an element is inserted, all successive elements move forwards to create a gap.

For some array types, if an element expands or shrinks in size, successive elements may need to shift forwards or backwards too.

Array

Can store 0…*unlimited* integers of range -2^{63} to $2^{63} - 1$ (same as signed 64-bit integers).

The bitwidth used to store the integers is selected automatically to use as little space as possible. All integers have the same bitwidth which is either 0, 1, 2, 4, 8, 16, 32 or 64. So, if the largest integer is 2, then 2 bits are used:

01	00	10

If we append the integer 7, all integers expand to 4 bits (expensive operation):

0001	0000	0010	0111

After deletion of second integer, successive items move backwards (expensive operation!):

0001	0010	0111

ArrayString

Can store 0…*unlimited* strings of length 0…63 bytes.

Strings are stored in concecutive fixed-width blocks. All blocks have the same length which is either 0, 4, 8, 16, 32 or 64 bytes and each string is 0-terminated and 0-padded:

hello000	**foo**00000	**foobar**00

If we modify the middle string to something longer, all blocks are expanded (expensive operation):

hello00000000000	**longstring**000000	**foobar**0000000000

If we delete it, all successive items are shifted backwards (expensive operation):

hello00000000000	**foobar**0000000000

ArrayStringLong

Can store 0…*unlimited* strings of length 0…*unlimited*.

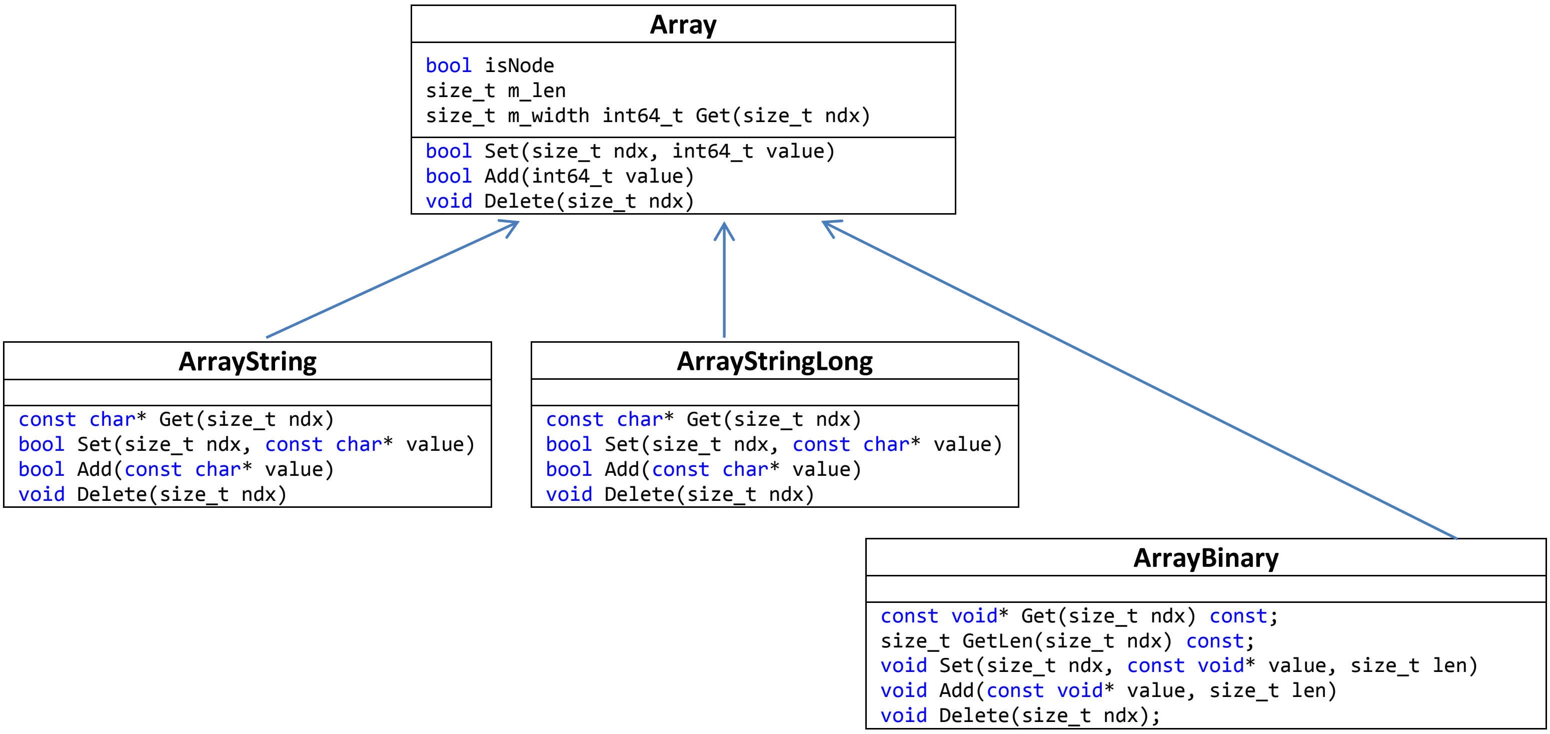
Strings are 0-terminated and stored concecutively with no padding:

Hello0	**The quick brown fox jumped over the lazy dog**0	**Hans Dampf hat dampf wie hansdampf**0

ArrayBinary

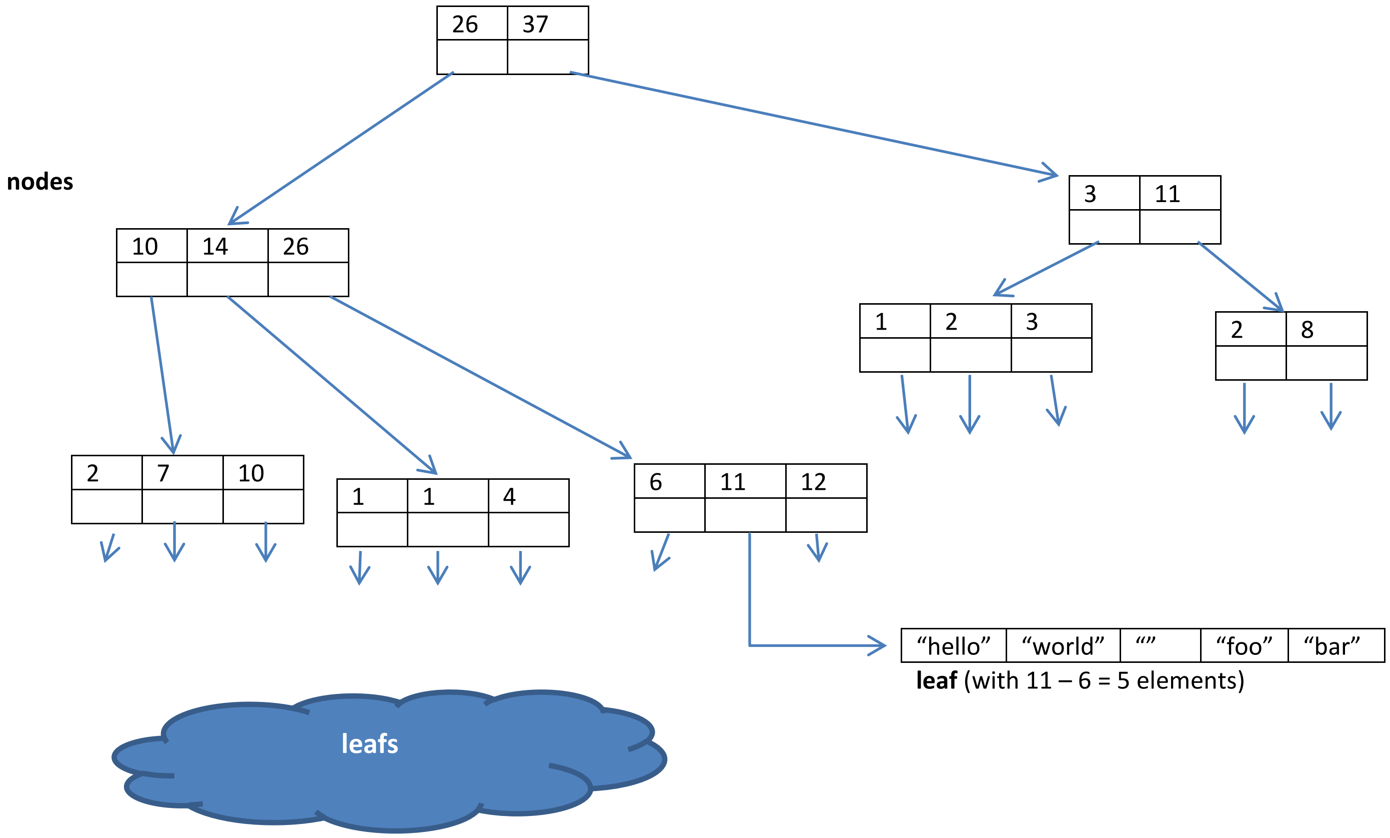
Can store 0…*unlimited* binary blocks of length 0…*unlimited*.

The different types of arrays **inerit from Array** which handles memory allocation, constructors and other common things. Array also contains information on wether it's a leaf or part of a node in the B-tree described later.



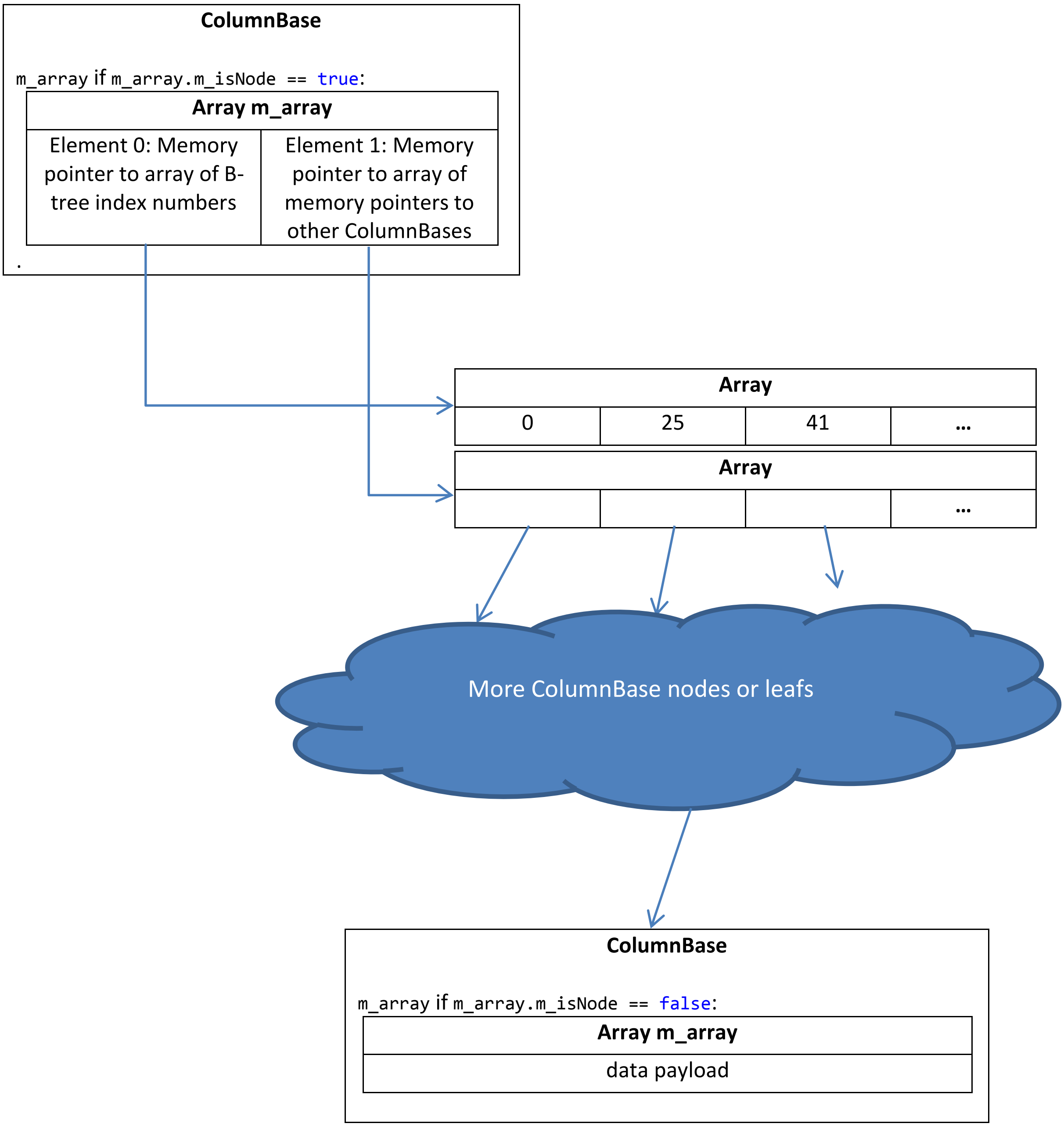
Because deletion and insertion in an array is expensive, we split data into multiple arrays, each containing at most MAX_LIST_SIZE elements - that way at most MAX_LIST_SIZE elements need to be moved in memory after each operation. The arrays are stored in a B-tree structure that allows fast lookup of a value by its index number.

A B-tree consists of nodes and leafs. A **node** consists of two arrays: One containing index offsets (upper array) and one containing pointers to other nodes or leafs (lower array). A **leaf** consists of one array containing data payload.



ColumnBase (Column.c / Column.h)

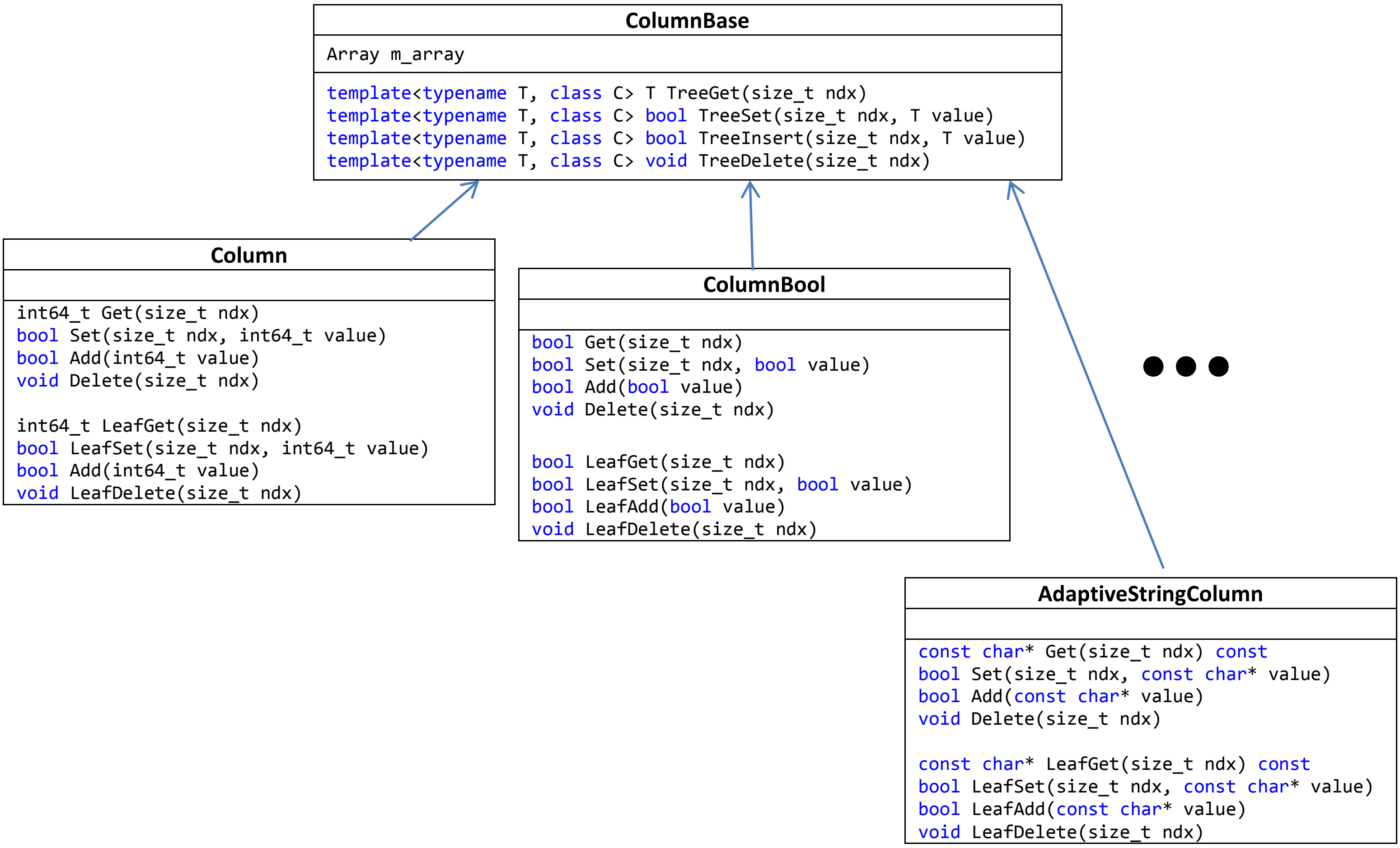
Each node and leaf is of the type ColumnBase which contains an array named m_array. If `m_array.m_isNode == true` it's a node, and if `m_array.m_isNode == false` it's a leaf containing data payload.



Columns

For each TightDB data type there exists a **subclass** of ColumnBase:

Column class name	Data type	Array type used for storage
`Column`	64-bit integers	Array
`AdaptiveStringColumn`	Strings of length 0…*unlimited*	ArrayString or ArrayStringLong (switches automatically)
`ColumnBinary`	Binary blocks of length 0…*unlimited*	ArrayBinary
`ColumnBool`	Boolean values	Array
`ColumnDate`	Dates	Array



The ColumnBase class offers templated B-tree functions for any data type and builds/maintains a B-tree. The subclass must offer non-templated wrappers for a public API and must also provide a non-templated “Leaf” function that returns leaf payload to ColumnBase.

Let’s look at a simple example of fetching the string “foo” in the above sample tree:

```
                                      main () {
                                            AdaptiveStringColumn c;
                                            ...
                                            char *s = c.Get(23);
                                      }

                                      const char* AdaptiveStringColumn::Get(size_t ndx) const {
                          ndx = 23          return TreeGet<const char*, AdaptiveStringColumn>(ndx);
                                      }

                                      template<typename T, class C> T ColumnBase::TreeGet(size_t ndx) const {
                                            if (IsNode()) {
                                                  // Get subnode table
                                                  const Array offsets = NodeGetOffsets(); // Element 0 of m_array
                                                  const Array refs = NodeGetRefs();       // Element 1 of m_array

                                                  // Find the subnode containing the item
ndx = 23, 0, 2, 1 for each recursion              const size_t node_ndx = offsets.FindPos(ndx);

                                                  // Calc index in subnode
                                                  const size_t offset = node_ndx ? (size_t)offsets.Get(node_ndx-1) : 0;
                                                  const size_t local_ndx = ndx - offset;

                                                  // Get item
                                                  const C target = GetColumnFromRef<C>(refs, node_ndx);
                                                  return target.TreeGet<T,C>(local_ndx);
                                            }
                                            else {
                                                  return static_cast<const C*>(this)->LeafGet(ndx);
                    recursive call          }
                                      }

                          ndx == 3    const char* AdaptiveStringColumn::LeafGet(size_t ndx) const {
                                            if (IsLongStrings()) {
                                                  return ((ArrayStringLong*)m_array)->Get(ndx);
                                            }
                                            else {
                                                  return ((ArrayString*)m_array)->Get(ndx);
                                            }
                                      }
```

So, to implement a new TightDB data type, you must:

1. Create a suited Array type or re-use an existing, supporting Get(), Set(), etc.
2. Create a column type that offers simple Get(), Set(), etc. wrappers around ColumnBase TreeGet(), TreeSet(), etc.
3. Create simple LeafGet(), LeafGet(), etc. wrappers around your Array::Get(), Array::Set(), etc. functions that ColumnBase can call.

Array class

Array	
`bool m_hasRefs;`	
`unsigned char* m_data;`	
`size_t m_len;`	
`size_t m_width;`	
`size_t m_ref;`	
`size_t m_capacity;`	
`bool m_isNode;`	

If you want to store persistent data (data which must be saved between program restarts, reboots, etc) you must use the Array class.

Persistent data is, for example:

- All user data payload
- Entire B-tree (nodes, leafs, references, index numbers - everything)
- Database structure (table names in a group, column names and types in each table, etc)