Exploring Code Size and Footprint, and In-Memory Database Techniques to Minimize Footprint

Are Either Relevant to Embedded Systems in the IoT Age?

Abstract: The terms ‘code size’ and ‘footprint’ are often used interchangeably. But they are not the same; code size is a subset of footprint. This paper will explain the differentiation and relevance, then proceed to describe some of the techniques employed within *e*X*treme*DB® to minimize footprint.

McObject LLC
33309 1st Way S.
A208
Federal Way, WA 98003

Phone: 425-888-8505
E-mail: info@mcobject.com
www.mcobject.com

1. Are ‘footprint’ and ‘code size’ the same thing? 2. And, is either relevant in the current age?

1. No. Code size is a subset of footprint. In the context of an in-memory database system, ‘footprint’ consists of:

Code size
Stack memory used
DRAM used for anything other than raw data

With respect to code size, we must have an all-encompassing view. We’re not the only embedded database vendor that boasts about a ‘small footprint’. But, *eXtreme*DB might be the only embedded database that has **no external dependencies**. For purposes of this discussion, it means that using the *eXtreme*DB core in-memory database does not cause the application to link in the C run-time library for things like malloc/free, string operations, etc. A claim of ‘small footprint’ is hollow when the application is forced to link a megabyte of the C run-time library.

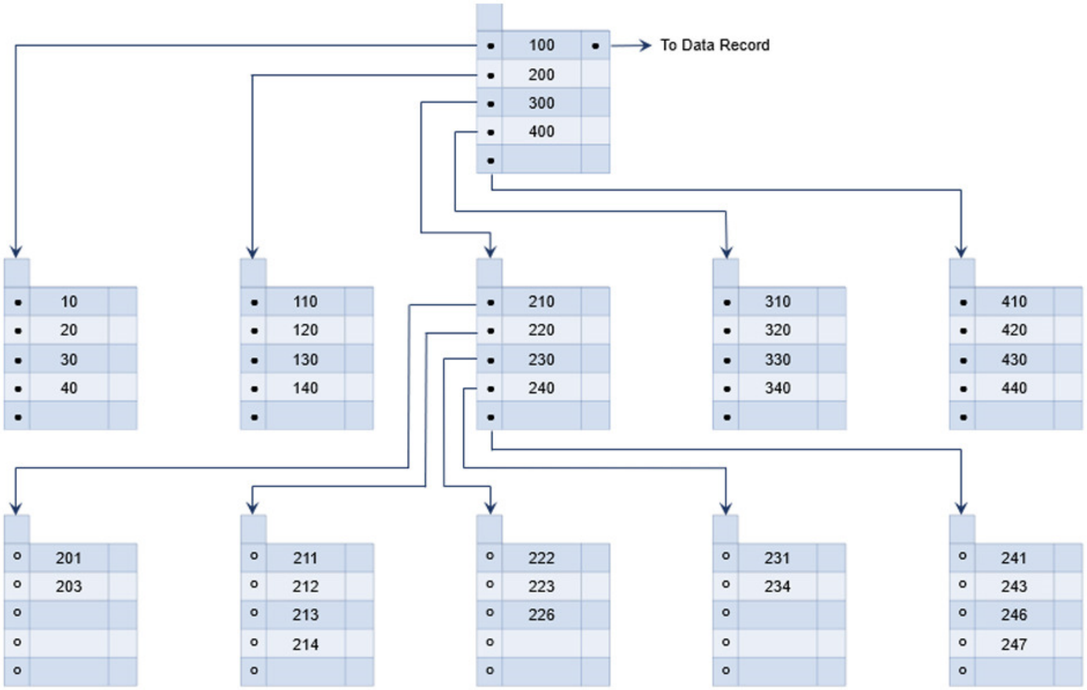
2. Yes, it’s relevant. Size matters. We can impute two things from a small code size: The function call depth and the number of CPU instructions for any given operation. A core embedded database engine with a code size of 150KB is going to require fewer function calls, and each function call is going to require fewer CPU instructions than another embedded database engine with a code size of 1.5MB. Whether you’re executing on a 200 mHz ARM processor or a 3 gHz Intel SkyLake processor, a database lookup that requires 100 CPU cycles is 10X faster than a database lookup that requires 1000 processor cycles because the code is that bloated.

Stack size in embedded/real-time operating systems is quite limited. For example, the default stack size with VxWorks is just 20KB. Unlike Windows and Linux, the stack does not grow dynamically. If the stack size is exceeded, the program fails. This ties directly back to function call depth, but also influences how much, and how, information is passed between functions, which is a consideration that must be considered when the database system is designed. This is a key differentiator between an embedded database system that was written for embedded systems, and an embedded database system that is merely capable of running on embedded systems. In *eXtreme*DB, information relied on by two or more functions in the call stack is sometimes put in our private heap and passed by reference, but never passed by value. In addition, indeterminate recursion is avoided altogether.

It should be obvious, but for an in-memory database, DRAM ***is*** storage space. Any memory that is used for anything other than raw data limits the amount of raw data that can be stored in any given amount of DRAM. In addition to storing raw data, memory is used for meta data (data about the data, the most common example being the database dictionary), index structures such as hash tables and b-trees, and database internals like transaction buffers, and handles of various types. Let’s use some real numbers to illustrate: suppose that in-memory database ‘A’ imposes 100% overhead on the raw data. That means that 20MB of DRAM will be required to store 10MB of raw data. In-memory database ‘B’ only imposes 30% overhead, so it only requires 13MB to store the same 10MB of raw data. In an embedded system like a portable media player that only has 32MB of memory to begin with, that’s a meaningful difference!

*eXtreme*DB is in-memory database ‘B’.

*eXtreme*DB employs a number of techniques to minimize the overhead. The first, and biggest, influencer is index types and how indexes are organized. *eXtreme*DB offers a wider variety of index types than other database systems, but the most commonly used are hash and b-tree indexes. By their nature, hash indexes use less memory (and are faster) than b-tree indexes. But, hash indexes have less utility: they don’t maintain sorted order or facilitate partial key searches. By far, the most common type of index in database systems is the b-tree index, which overcomes the limitations of the hash index, and offers decent Log$_2$*N* performance. To understand the advantages and, for purposes of this discussion, disadvantages of a b-tree you need to have at least a high-level understanding of their organization.



A b-tree is an upside-down tree, with the ‘root’ node at the top. Each node has a number of slots where the number of slots is a function of the size of the node (in bytes) and the size of the ‘payload’. Simplistically, number-of-slots = node size / payload. Each slot contains

(1) a pointer to a node with key values that precede the key value in this slot
(2) this key value
(3) a pointer to the actual data record that is being indexed.

As you can see in the illustration, b-trees are normally partially empty (up to 45% empty), so there’s built-in space inefficiency. The key value exists in the slot so that database systems can implement an optimization called a ‘covered query’, which simply means that if column(s) queried are only the indexed column(s), then the query’s results can be obtained by only scanning the index; there is no need to follow the pointer to the data record and the amount of logical I/O is reduced by one-half. But, this optimization only makes sense for persistent databases, because of the speed of storage media. Following a pointer in memory to the data record takes just nanoseconds, whereas following a pointer to a track and sector on a solid state disk can be as low as 1 microsecond, and generally on the order of 10 milliseconds for a hard disk drive. The use of precious DRAM to store redundant key values is unjustified, in most cases. By default, *eXtreme*DB b-tree indexes do not store redundant key values in b-tree node slots. This leads to another footprint-saving advantage: Index slot sizes are constant because the slot only contains a reference to the indexed data, not the data itself. So, regardless of whether an index is over a 2-byte short integer, or a 64-byte character field, the slot size is the same, which simplifies the code.

The size of b-tree nodes is normally a multiple of the disk blocking factor to maximize I/O efficiency. In other words, if the disk block size is 4096 bytes, then the b-tree node size is e.g. 4096, 8192, or 16384 bytes. It makes no sense to have an index node size of 1024 knowing that the disk and file system are going to perform I/O in units of 4096; 3072 bytes of that I/O would be for no productive purpose. In addition to maximizing I/O efficiency, a large(-ish) node size keeps the b-tree more shallow. For a persistent database, each level that we need to descend to find the search value equates to a logical I/O. A b-tree that is five levels deep will require >3 probes into the b-tree on average. So, having large node sizes means having a shallower tree which means fewer logical I/O which means better performance. However, none of this calculus holds up for b-trees in memory. Having large nodes means having more empty slots which means more memory consumed for no productive purpose. And, given the fast access of DRAM, having an average number of probes of ~4 is not meaningfully different than an average number of ~3 probes, but having fewer empty slots can translate to substantially less memory consumption. Also, there's no reason to harmonize b-tree node size with anything because DRAM is not a block device. So, in-memory b-tree nodes are usually small, for example a couple of hundred bytes. Finally, having a fixed slot size by not including the indexed data in the slot usually means having smaller slots, which translates to more slots per node which translates to a shallower tree.

B-tree indexes are incredibly flexible. We can use them for =, >, <, >=, <=, range searches, sorted order retrieval and partial key searches. But, sometimes, we only need to look up one exact record, e.g. find the artist 'Elton John'. For this, a hash index is far superior. First, there's minimal wasted space; it's a table that is sized based on the anticipated number of entries. Second, it's a table, not a tree, so it has an inherent performance advantage. Instead of walking the tree nodes, a hash value is computed (based on some algorithm) which translates a key value to hash table entry (offset). After a few CPU cycles to calculate the hash, the DBMS steps right into the hash table at that offset and follows the pointer to the data record. Hash indexes save memory and outperform b-tree indexes, most of the time (every rule has exceptions).

The second biggest influencer in *e*X*treme*DB's frugal use of memory is the layout of the data. In most databases systems, when a table is defined (e.g. through an SQL CREATE TABLE statement), the data is stored exactly how it was defined. For example:

```
CREATE TABLE example (
        Column1         char(1),
        Column2         int,
        Column3         char(2),
        Column4         float(4)
);
```

Would be laid out in memory (and on storage, in the case of a persistent database) as

BYTE	CONTENT
1	Column1
2	padding to align the int on a 2-byte boundary
3,4	Column2
5,6	Column3
7	padding to align the float on a 4-byte boundary
8,9,10,11	Column4

Each stored row would waste two bytes of memory due to the padding. If there are 1 million rows, that's 2 million wasted bytes which is, again, meaningful in a device that only has 32MB of memory to begin with. But, this is also meaningful in so-called Big Data: 100 million rows laid out as above would consume 1,100,000,000 (1.1 GB) of memory, of which 200 MB is padding.

A better solution is for the DBMS to rearrange the fields to eliminate the padding:

BYTE	CONTENT
1,2,3,4	Column4
5,6	Column2
7,8	Column3
9	Column1

The order of Column3 and Column1 is not actually important in this case. In the Big Data case, this layout would only require 900 MB of DRAM, or permit 22,222,222 more rows in the same 1.1 GB of DRAM as the non-aligned data layout.

Another significant influencer in *eXtreme*DB's frugal use of memory is it's use of proprietary purpose-specific memory managers. A memory manager like the C run-time's malloc is called a *list allocator*. Available memory is organized as a linked list (chain) of available blocks of memory. There's a pointer to the first block in the list and an integer that records what the size of that block is. That block has a pointer to the next block and an integer indicating the size of that block, and so on for every block of available memory. For a 32-bit system, that's 8 bytes of overhead for every free block and 16 bytes of overhead for a 64-bit system. Like a b-tree, a list allocator is very versatile, it can allocate blocks of memory of varying sizes and coalesce adjacent free blocks into a single larger free block. But, sometimes we don't need that flexibility and conserving memory would be more important. One example is managing database pages (that are used to store data and index nodes). Per the previous discussion, the size of these objects is fixed, so we don't need the versatility to allocate random size pieces of memory. A *block allocator* is a better fit. A block allocator subdivides a large block of memory into smaller fixed size blocks and like the list allocator maintains a list of free blocks. But the size of each free block is constant and known, so the amount of overhead is immediately cut in half. Another example is found in SQL engines. When a SQL statement is received, it needs to be parsed, then optimized, then executed and the result set created. Each of these steps requires amounts of memory that can vary from one SQL statement to the next, so we want the flexibility of an allocator, but the nature of the allocations is that all of the memory is allocated in the first phase, and then all of it is released in the next phase. In other words, allocations and frees are not intermixed. For this pattern, a *stack allocator* fits. A stack allocator is given a block of memory and keeps a pointer to the first byte. If 10 bytes are allocated, the pointer is advanced 10 bytes, and so on for each subsequent allocation. When the SQL engine is done with each phase of the statement, the pointer is rewound to the start. There is virtually no overhead with this allocator; there's no list of pointers and no need to track the size of each allocated or freed block. Another pleasant byproduct of the stack allocator is performance; there may be hundreds of thousands of individual memory allocations while parsing, optimizing and executing a SQL statement, but a single 'free' rewinds the pointer and there is no need to free each prior allocation one-by-one. This also eliminates the potential for memory leaks, so there's a safety advantage, as well. Memory allocators are explored in more detail in this on-demand webinar.

In summary, code size and footprint are not the same thing. Both are still relevant. For an embedded database system, code size should refer only to the object code library size and the required external libraries (e.g. the C runtime). Smaller is better. Footprint incorporates code size, but also needs to factor in everything else that contributes to the overall memory consumption of the final solution. Especially in embedded/real-time resource constrained systems, and in-memory databases, footprint is an important consideration.