Code Size vs. Memory Footprint

10/01/2020

Written by Steve Graves

And Why They Matter

The terms "code size" and "footprint" are often used interchangeably. But they are not the same—code size is a subset of footprint. In this article, McObject CEO Steve Graves explains the difference and relevance. He then describes some techniques to minimize footprint.

Consider these two qu
size" the same thing?
age?

Answer 1: No. Code s
context of an in-memo
consists of:

Topics Covered

Tech Discussed

• 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." That said, eXtremeDB from McObject might be the only embedded database that has *no external dependencies*. For purposes of this discussion, it means that using the eXtremeDB core in-memory database does not cause the application to link in the C run-time library for things like `malloc/free, string` operations and so forth. A claim of "small footprint" is hollow when the application is forced to link a megabyte of the C run-time library.

Answer 2: Yes, these terms are relevant. Code size matters. We can ascribe two things to 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 200MHz Arm processor or a 3GHz Intel Sky Lake processor, a database lookup that requires 100 CPU cycles is 10× faster than a database lookup that requires 1,000 processor cycles.

STACK SIZE
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 that was written *for* embedded systems, and an embedded database that is merely capable of running *on* embedded systems. **Figure 1** illustrates the function call stack.

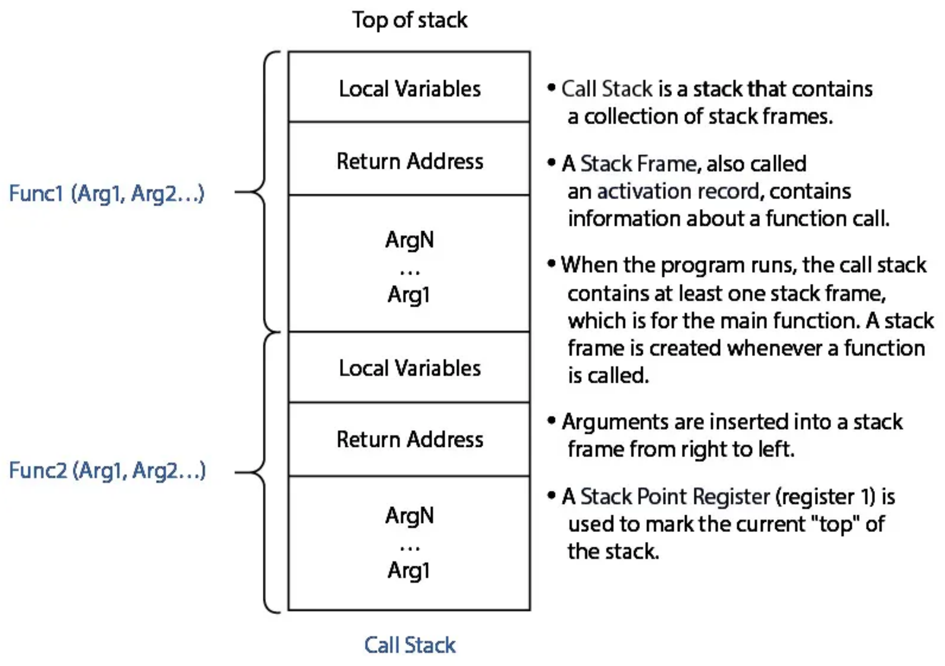
Top of stack
Func1 (Arg1, Arg2...)
Func2 (Arg1, Arg2...)
Local Variables
Return Address
ArgN
...
Arg1
Local Variables
Return Address
ArgN
...
Arg1
Call Stack
• Call Stack is a stack that contains a collection of stack frames.
• A Stack Frame, also called an activation record, contains information about a function call.
• When the program runs, the call stack contains at least one stack frame, which is for the main function. A stack frame is created whenever a function is called.
• Arguments are inserted into a stack frame from right to left.
• A Stack Point Register (register 1) is used to mark the current "top" of the stack.

FIGURE 1 – An illustration of the Call Stack function

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), indexes and database internals.

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 is a meaningful difference! eXtremeDB is an example of an in-memory database type "B" in this comparison. The relationship of code, static data, stack and heap is illustrated in **Figure 2**.

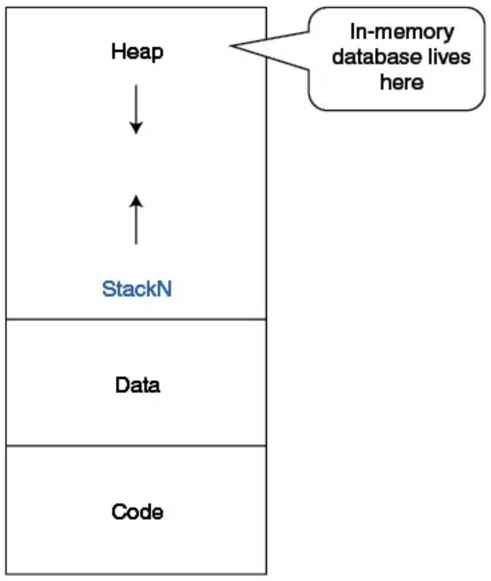

FIGURE 2 – This shows the relationship of the code, the static data, the stack and the heap.

For its part, eXtremeDB employs several techniques to minimize the overhead. The first—and biggest—influencer is index types and how indexes are organized. eXtremeDB 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. By far, the most common type of index is the b-tree index. To understand the disadvantages of a b-tree you need to have at least a high-level understanding of their organization.

A B-TREE INDEX

As shown in **Figure 3**, a b-tree is an upside-down tree, with the "root" node at the top. Each node has many 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, and (3) a pointer to the actual data record that is being indexed.

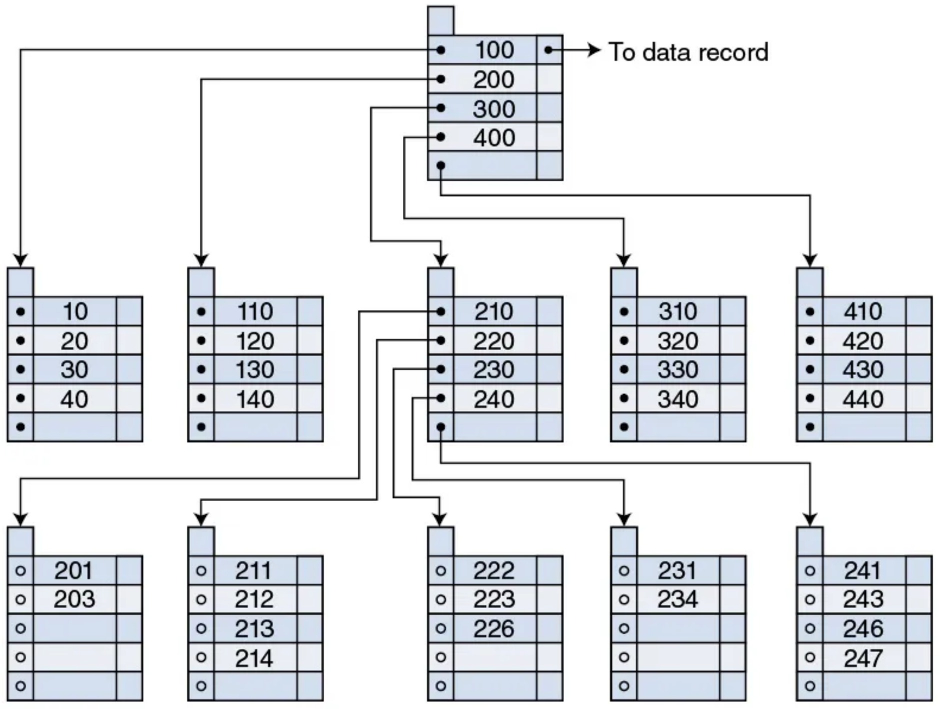

FIGURE 3 – A b-tree is structured like an upside-down tree, with the "root" node at the top. As you can see, b-trees are normally partially empty (up to 45% empty), so there's built-in space inefficiency.

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 the column(s) queries are only the indexed column(s), then the query results can be obtained by only scanning the key. 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 1µs, 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, eXtremeDB b-tree indexes do not store redundant key values in b-tree node slots.

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 4,096, then the b-tree node size is, for example 4,096, 8,192 or 16,384. 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 greater than 3 probes into the b-tree on average. So, having large node sizes means having a more shallow 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 approximately 4 is not meaningfully different than an average number of around 3 probes, but having fewer empty slots can translate to substantially less memory consumption. Also, there is 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.

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 —for example, find the artist "Elton John." For this, a hash index is far superior. First, there's minimal wasted space. It is a table that is sized based on the anticipated number of entries. Second, it is 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). The DBMS (database management system) 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).

DATA LAYOUT

The second biggest influencer in eXtremeDB's frugal use of memory is the layout of the data. In most databases systems, when a table is defined (for example, 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
[34] Column2
[56] Column3
[7] padding to align the float on a 4-byte boundary
[8901] Column4
```

Each stored row would waste two bytes of memory due to the padding. If there are 1 million rows, that is 2 million wasted bytes which is, again, meaningful in a device that only has 32MB of memory to begin with.

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

```
BYTE CONTENT
[1234] Column4
[56] Column2
[78] Column3
[9] Column1
```

The order of Column3 and Column1 is not important in this case.

Another significant influencer in eXtremeDB's frugal use of memory is its use of proprietary purpose-specific memory managers. A memory manager like the C run-time's `malloc` is called a "list" manager. Available memory is organized as a linked list (chain) of available blocks of memory. There is a pointer to the first block in the list and an integer that records the size of that block. 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 is 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 do not 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 do not 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 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.

SQL ENGINE EXAMPLE

Another example is found in SQL engines. When a SQL statement is received, it needs to be parsed, then optimized, then executed. 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 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 is a safety advantage as well.

In summary, code size and footprint are not the same thing. Both are still relevant. For an embedded database, code size should refer only to the object code library size. 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.

RESOURCES

McObject | www.mcobject.com

PUBLISHED IN CIRCUIT CELLAR MAGAZINE • OCTOBER 2020 #363 – Get a PDF of the issue