Resource Reduction in Q

July 15, 2024

Abstract

This document lists various techniques used to reduce Q's resource consumption (in terms of memory and disk usage).

1 Introduction

1.1 Motivation

Since Q is an extension of Lua (LuaJIT to be precise), it performs garbage collection. This allows it to free resources that are no longer needed.

However, there are two weaknesses with this approach.

1. Lua is unable to look inside a Vector to assess how much memory/disk it is using and therefore whether garbage collection should be triggered.
2. Even if it could, there are many situations where the Q programmer can use their knowledge of the computation to free resources earlier than they would have been freed had one relied solely on garbage collection. This is consistent with Q's overall design philosophy of creating hooks that allow the advanced programmer to gain efficiency at the cost of thinking harder.

1.2 Organization

We provide the Q programmer with three mechanisms to inform the Q runtime that a vector can be freed in total or in part. When using these mechanisms, we impose some constraints on how they can be used not for any intrinsic reason but because of my general belief that one should provide the system with as few degrees of freedom without hamstringing the Q programmer.

1. memo — Section 2
2. kill — Section 3
3. early free — Section 4

Name in LaTeX	In `q_cfg.lua`	C Type	C variable	Default
n_E		uint64_t	num_elements	
p		bool	is_persist	false
$c_{\min}$		uint32_t	min_chnk_idx	
$c_{\max}$		uint32_t	max_chnk_idx	
m	Y	bool	is_memo	false
n_M	Y	uint32_t	memo_len	0
k	Y	bool	is_killable	false
n_K	Y	uint32_t	num_kill_ignore	0
f	Y	bool	is_freeable	false
n_F	Y	uint32_t	num_free_ignore	0

Table 1: Data Structures for Early Resource Allocation

1.3 Data Structures

See Table 1

1.4 Default Values

Default values for properties marked with a Y in Table 1 are set in `~/Q/UTILS/lua/qcfg.lua` They can be changed using `qcfg._modify()`

1.5 Persistence

A vector will **not** be persisted unless $m = k = f =$ false.

1.6 When can changes be made

1.6.1 memo

m can be set to true only if

1. $n_E = 0$
2. $f =$ false
3. $p =$ false

1.6.2 kill

k can be set to true only if

1. $n_E = 0$
2. $p =$ false

1.6.3 free

f can be set to true only if

1. $n_E = 0$
2. $p =$ false
3. $m =$ false

2 Memo

If we do not "memo" the vector, then it holds on to all chunks created. We use memo to indicate the maximum number of chunks the vector should hold on to. For example, say we set the memo len to 2. When we create the first chunk, the vector holds on to chunk $\{0\}$. When we create the second chunk, the vector holds on to chunks $\{0, 1\}$. When we create the third chunk, the vector discards the 0^{th} chunk and holds on to chunks $\{1, 2\}$

2.1 Invariants

Invariant 1 $m = \text{false} \Rightarrow n_M = 0$

Invariant 2 $m = \text{true} \Rightarrow n_M > 0$

For a Vector that has been memo'd, the number of chunks cannot exceed n_M. Hence, Invariant 3

Invariant 3 $m = \text{true} \Rightarrow c_{\max} - c_{\min} + 1 \leq n_M$

3 Kill-able

Killable was invented to handle the case when we know that a vector will no longer be needed. For example, assume $x \leftarrow y + z$ and we know that once x has been computed, there is no further need for y. In that case, the creator of x will issue a kill signal to y. If y was marked killable, then it is deleted and all its resources freed.

To complicate matters, when we set a vector as killable, we set the number of lives that it has. In the above example, assume that y was needed not just for the creation of x but also for the creation of $w \leftarrow \sum y$. In that case, we would set $n_K(y) \leftarrow 1$. Note that the creator of x and the creator of w both issue kill signals to y.

When y receives the first kill signal, it is ignored because $n_K(y) \neq 0$. However, we decrement $n_K(y)$ by 1 , setting it to 0 in this example. When y receives the second kill signal, it is not ignored because now $n_K(y) = 0$ and y is deleted.

As usual, if $k(y) =$ false, then kill signals received by y are ignored.

3.1 Invariants

Invariant 4 $k = \text{false} \Rightarrow n_K = 0$

Questions

1. Can you kill a vector that is not yet eov?

4 Free-able

There are situations (e.g., the `where` operator) where we do not in advance how many chunks we need to remember. However, we do know that if the vector receives a "free" signal, then it can delete all but the most recent chunk. We require the `early_free()` call to provide a chunk index. So, `early_free(`i`)` means that chunks $\{0, 1, 2 \ldots i-1\}$ will be deleted.

Unfortunately, the situation gets more complicated, akin to Section 3, because a vector may have more than one consumer. Setting $f \leftarrow \text{true}$ requires us to set $n_F \geq 0$. Assume $n_F > 1$. When a chunk is created, we assign it a "free-life" of n_F. When, $x : (\text{early\_free}i)$ is invoked, each chunk whose index is $< i$ is examined. If its free life is 0, it is deleted. If its free life is > 0, then its free life is decremented by 1.

As usual, if $f(y) = \text{false}$, then early free signals received by y are ignored.

You cannot use memo and freeable at the same time. Hence, Invariants 5, 6.

Invariant 5 $m = \text{true} \Rightarrow f = \text{false}$

Invariant 6 $f = \text{true} \Rightarrow m = \text{false}$