Brief Announcement: Large-Scale Multimaps

Michael T. Goodrich
Dept. of Computer Science
University of California, Irvine
goodrich(at)acm.org

Michael Mitzenmacher
Dept. of Computer Science
Harvard University
michaelm(at)eecs.harvard.edu

ABSTRACT

Many data structures support dictionaries, also known as maps or associative arrays, which store and manage a set of key-value pairs. A ***multimap*** is generalization that allows multiple values to be associated with the same key. We study how multimaps can be implemented efficiently online in external memory frameworks, with constant expected I/O. The key technique used to achieve our results is a combination of cuckoo hashing using buckets that hold multiple items with a multiqueue implementation to cope with varying numbers of values per key.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Algorithms, Theory

Keywords

Multimap, inverted index, cuckoo hashing, multiqueue.

1. INTRODUCTION

A ***multimap*** is a simple abstract data type (ADT) that generalizes the the map ADT to support key-value associations in a way that allows multiple values to be associated with the same key. Specifically, it is a dynamic container, C, of key-value pairs, which we call ***items***, supporting (at least) the following operations:

- insert(k, v): insert the key-value pair, (k, v). This operation allows for there to be existing key-value pairs having the same key as k, but we assume w.l.o.g. that the particular key-value pair (k, v) is itself not already present in C.
- isMember(k, v): return true if and only if the key-value pair, (k, v), is present in C.
- remove(k, v): remove the key-value pair, (k, v), from C. This operation returns an error condition if (k, v) is not currently in C.
- findAll(k): return the set of all key-value pairs in C having key equal to k.
- removeAll(k): remove from C all key-value pairs having key equal to k.

SPAA'11, June 4–6, 2011, San Jose, California, USA.
ACM 978-1-4503-0743-7/11/06.

Surprisingly, we are not familiar with any previous discussion of this specific abstract data type in the theoretical algorithms and data structures literature. Nevertheless, abstract data types equivalent to the above ADT, as well as multimap implementations, are included in the C++ Standard Template Library (STL), Guava–the Google Java Collections Library, and the Apache Commons Collection 3.2.1 API. The existence of these implementations provides empirical evidence for the utility of this abstract data type.

One of the primary motivations for studying the multimap ADT is that associative data in the real world can exhibit significant non-uniformities with respect to the relationships between keys and values. For example, many real-world data sets follow a power law with respect to data frequencies indexed by rank. Specifically, in natural language documents, the frequency of the word of rank j is predicted to be roughly proportional to j^{-s} for some parameters s. Thus, if we wished to construct a data structure to retrieve all instances of any query word in such a corpus, subject to insertions and deletions of documents, then we could use a multimap, but would require one that could handle large skews in the number of values per key. In this case, the multimap could be viewed as providing a dynamic functionality for a classic static data structure, known as an ***inverted file*** or ***inverted index*** (e.g., see Knuth [3]). Such data structures are often used in modern search engines (e.g., see Zobel and Moffat [9]). Dynamic inverted indexes have been studied in the past, but generally from a systems viewpoint rather than a theoretical one. (See, e.g., [4, 5], and references therein.)

Our work utilizes a variation on cuckoo hash tables. We assume the reader has some familiarity with such hash tables, as originally presented by Pagh and Rodler [6]. (A general description can be found on Wikipedia at `http://en.wikipedia.org/wiki/Cuckoo_hashing`.) We describe an external-memory implementation of the multimap ADT, based on the standard two-level I/O model (e.g., see [1, 8]). We also have a parallel algorithm abstracted using the bulk synchronous parallel (BSP) model [7], which we do not describe due to lack of space. We support an online implementation where each operation must be completely finished executing prior to our beginning execution of any subsequent operations. The bounds we achieve are shown in Table 1.

Our constructions are based on the combination of external-memory cuckoo hash tables and multiqueues. We show that external-memory cuckoo hashing supports a cuckoo-type method for insertions that can be implemented in a way that allows us to prove that only an expected constant num-

Method	Amortized I/O Performance
insert(k, v)	$\bar{O}(1)$
isMember(k, v)	$O(1)$
remove(k, v)	$O(1)$
findAll(k)	$O(1 + n_k/B)$
removeAll(k)	$O(1)$

Table 1: Performance bounds for our multimap implementation. We use $\bar{O}(*)$ to denote an expected bound; B to denote the block size; N to denote the number of key-value pairs; and n_k to denote the number of key-value pairs with key equal to k.

ber of I/Os are needed to find a place where each new item can be placed. We then show that this performance can be combined with amortized expected constant I/O complexity for multiqueues to design a multimap implementation that has constant amortized worst-case or expected I/O performance for most methods. Our methods imply that one can maintain an inverted file in external memory so as to support a constant amortized expected number of I/Os for insertions and worst-case constant amortized I/Os for lookups and item removal.

2. BRIEF SKETCH

For our external-memory cuckoo hash table, each bucket can store up to B items, where B defines our block size and is not necessarily a constant. Formally, let $\mathcal{T} = (T_0, T_1)$ be a cuckoo hash table such that each T_i consists of $\gamma n/2$ buckets, where each bucket stores a block of size B, with $n = N/B$. Of particular interest is when $\gamma = 1 + \epsilon$ for some (small) $\epsilon > 0$, so that space overhead of the hash table is only an ϵ factor over the minimum possible. The items in $\mathcal{T}$ are indexed by keys and stored in one of two locations, $T_0[h_0(k)]$ or $T_1[h_1(k)]$, where h_0 and h_1 are random hash functions.

We modify previous analyses of two-choice cuckoo hashing with multiple items per bucket from [2] by splitting the items into sub-buckets to show that we can stay within a $1+\epsilon$ factor of the total space required for all elements while maintaining an expected $\log(1/\epsilon)^{O(\log\log(1/\epsilon))}$ insertion time, using a breadth first search approach. As noted in [2], a more practical approach is to use *random walk cuckoo hashing* in place of breadth first search cuckoo hashing, but it is not known if there is a random walk cuckoo hashing scheme using $(1+\epsilon)N$ total space for N items, two bucket choices, and multiple items per bucket that similarly achieves expected constant insertion time and logarithmic insertion time with high probability.

To implement the multimap ADT, we begin with a primary structure that is an external-memory cuckoo hash table storing just the set of keys. In particular, each record $R(k)$ in $\mathcal{T}$ is associated with a specific key k and holds the following fields: the key k; the number n_k of key-value pairs in C with key equal to k; and a pointer p_k to a block X in a secondary table, $\mathcal{S}$, that stores items in C with key equal to k. If $n_k < B$, then X stores all the items with key equal to k (plus possibly some items with keys not equal to k). Otherwise, if $n_k \geq B$, then p_k points to a ***first*** block of items with key equal to k, with the other blocks of such items being stored elsewhere in $\mathcal{S}$.

This secondary storage is an external-memory data structure we call a ***multiqueue***. It maintains a set $\mathcal{Q}$ of queues in external memory. The ***header*** pointers for these queues are stored in an array $\mathcal{T}$, which in our external-memory multimap construction is the external-memory cuckoo hash table described above. For any queue Q, we wish to support the following operations: enqueue(x, H) adds the element x to Q given a pointer to its header H; remove(x) removes x from Q; and isMember(x): determine whether x is in some queue Q. In addition, we wish to maintain all these queues in a space-efficient manner, so that the total storage is proportional to their total size. To enable this efficiency, we store all the blocks used for queue elements in a secondary table, $\mathcal{S}$, of blocks of size B each. Thus, each header record H in $\mathcal{T}$ points to a block in $\mathcal{S}$. In addition, we maintain a second cuckoo table, D, which uses entire key-value pairs as its keys. For each one, it provides a pointer to the block in $\mathcal{S}$ that stores this key-value pair (the data structure, D, is what allows us to perform fast removals).

Our intent is to store each queue Q as a doubly-linked list of blocks from $\mathcal{S}$. Unfortunately, some queues in $\mathcal{Q}$ are too small to deserve an entire block in S dedicated to storing their elements. So small queues must share their first block of storage with other small queues until they are large enough to deserve dedicated storage blocks. Managing the small and large queues with dynamic insertions and deletions is the further challenge in our construction, which we give in the full version of this paper[1].

3. REFERENCES

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, 1988.

[2] M. Dietzfelbinger and C. Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, 380:47–68, 2007.

[3] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

[4] N. Lester, A. Moffat, and J. Zobel. Efficient online index construction for text databases. *ACM Trans. Database Syst.*, 33:19:1–19:33, September 2008.

[5] N. Lester, J. Zobel, and H. Williams. Efficient online index maintenance for contiguous inverted lists. *Inf. Processing & Management*, 42(4):916–933, 2006.

[6] R. Pagh and F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 52:122–144, 2004.

[7] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[8] J. S. Vitter. External sorting and permuting. In M.-Y. Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008.

[9] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38, July 2006.

[1]Goodrich was supported in part by the NSF under grants 0724806, 0713046, and 0847968, and by the ONR under MURI grant N00014-08-1-1015. Mitzenmacher was supported in part by the NSF under grants 0915922 and 0964473.