Using Judy Arrays: Maximizing Scalability and Performance

John Abegg
john_abegg@hp.com
Hewlett-Packard Company
3404 East Harmony Road
Fort Collins, Colorado USA 80528 -9599
970-669-3883
April 20, 2001

What is Judy?

Judy is a C library that provides a state-of-the-art core technology that implements a sparse dynamic array. Judy arrays are declared simply with a null pointer. A Judy array consumes memory only when it is populated, yet can grow to take advantage of all available memory if desired.

Judy's key benefits are scalability, high performance, and memory efficiency. A Judy array is extensible and can scale up to a very large number of elements, bounded only by machine memory. Since Judy is designed as an unbounded array, the size of a Judy array is not pre-allocated but grows dynamically according to program requirements.

Judy combines scalability with ease of use. The Judy API is accessed with simple insert, retrieve, and delete calls that do not require extensive programming. Tuning and configuring are not required (in fact not even possible). In addition, sort, search, count, and sequential access capabilities are built into Judy's design.

Judy can be used whenever a developer needs dynamically sized arrays, associative arrays or a simple-to-use interface that requires *no* rework for expansion or contraction.

Judy can replace many common data structures, such as arrays, sparse arrays, hash tables, B-trees, binary trees, linear lists, skiplists, sort and search algorithms, and counting functions.

Judy and High Availability

As stated, the most significant benefit of the Judy technology is its ability to scale to very large data sets. As the data requirements for an application grow, Judy expands dynamically. This feature becomes significant in applications that use lots of data. Depending on the application, that data may grow to become quite sizable. Judy arrays scale as needed, so there is no need to stop an application to "re-compile" for larger data sets. Nor is there any need to "re-tune" an application so that it continues to perform well as data requirements grow.

Target Customer

Judy is designed for software developers creating C language applications.

Scalability

Judy has been used successfully for arrays with as few as one element (actually even zero) to arrays with billions of elements. Judy's is designed to support machines with memory much larger than is even currently possible (for example, multi-petabyte memories) since the technology is self-adapting.

Judy Design Goals

Judy is designed to replace many common data structures, such as arrays, sparse arrays, hash tables, B-trees, binary trees, linear lists, skiplists, and counting functions. Judy functions combine dynamic arrays with innovative memory management and retrieval.

Judy is implemented as a tree that dynamically optimizes each node based on the population under that node. Judy is also cache efficient, it optimizes memory access to be more efficient than other algorithms that ignore memory caching. As a result, Judy exhibits better than $O(\log_{256} N)$ retrieval behavior. Like other algorithms, such as B-trees but notably excluding hashing, Judy keeps indexes in sorted order.

Judy also includes a powerful counting feature that returns the number of elements (index/value pairs) between any two indexes or counts to any ordinal (position) within an array. For example, Judy can store the first million prime numbers. Then it can efficiently determine how many prime numbers exist between any random pair of numbers (prime or not). Because Judy maintains counts internally, the count function call can return an answer very quickly.

When to Use Judy

Designing programs with a sparse dynamic array paradigm is one of the most powerful uses of Judy. Use Judy when your program requires:

- The ability to scale effortlessly for future growth.
- Fast search and retrieval.
- Fast counting and sorting.
- Excellent performance across various types of index sets: sequential, periodic, clustered, and random.
- Memory efficiency: a low byte-per-index ratio for any array population.
- Nearly linear performance from very small to very large populations.

OS Support

Judy is currently supported on HP-UX 11i. Judy is available at no cost for HP-UX 11i users. The code can also be downloaded from the Judy web site (see end of paper). Both archive and shared libraries are provided for PA 1.1 and PA 2.0 hardware, for 32- and 64-bit indexes. We can port Judy to other operating systems and hardware architectures very easily. We plan to provide Judy support in the near future for HP-UX and Linux on the IA-64 platform.

Types of Judy Arrays

There are three types of Judy arrays:

Judy1 functions - "unbounded" dynamic bit arrays with arbitrary indexes, counting, and sorting features.

JudyL functions - "unbounded" dynamic word arrays with arbitrary indexes, counting, and sorting features.

JudySL functions - "unbounded" dynamic word arrays with arbitrary string indexes and sorting features.

Judy1 Functions

Judy1Test	Indicate whether a word-sized index has its bit set in the array.
Judy1Set	Set the bit for a given index.
Judy1Unset	Clear the bit for a given index.
Judy1First	Locate the first index with a bit set in the array (starting with the passed index).
Judy1Next	Locate the next index with a bit set in the array (starting above the passed index).
Judy1Last	Locate the last index with a bit set in the array (starting from the passed index).
Judy1Prev	Locate the previous index with a bit set in the array (starting below the passed index).
Judy1Count	Return the count of indexes with bits set between the specified indexes, including the specified indexes themselves.
Judy1ByCount	Locate the Nth index (N=count) with a bit set in the bit array.
Judy1FreeArray	Free the entire bit array.

JudyL Functions

JudyLGet	Retrieve a word-sized value from a JudyL array based on its word-sized index.
JudyLIns	Insert an index/value pair into a JudyL array.
JudyLDel	Delete an index/value pair.
JudyLFirst	Locate the first index stored in the array (starting with the passed index).
JudyLNext	Locate the next index stored in the array (starting above the passed index).
JudyLLast	Locate the last index stored in the array (starting from the passed index).
JudyLPrev	Locate the previous index stored in the array (starting below the passed index).
JudyLCount	Return the count of stored indexes between the specified indexes, including the specified indexes themselves.
JudyLByCount	Locate the Nth index (N=count) that is stored in the JudyL array.
JudyLFreeArray	Free the entire JudyL array.

JudySL Functions

JudySLGet	Retrieve a value from a JudySL array using a string as an index.
JudySLIns	Insert a value into a JudySL array using a string as an index.
JudySLDel	Delete an index (string/value pair).
JudySLFirst	Locate the first index stored in the array (starting with the passed index).
JudySLNext	Locate the next index stored in the array (starting above the passed index); continue a neighbor search.
JudySLLast	Locate the last index stored in the array (starting with the passed index).
JudySLPrev	Locate the previous index stored in the array (starting below the passed index); continue a neighbor search.
JudySLFreeArray	Free the entire JudySL array.

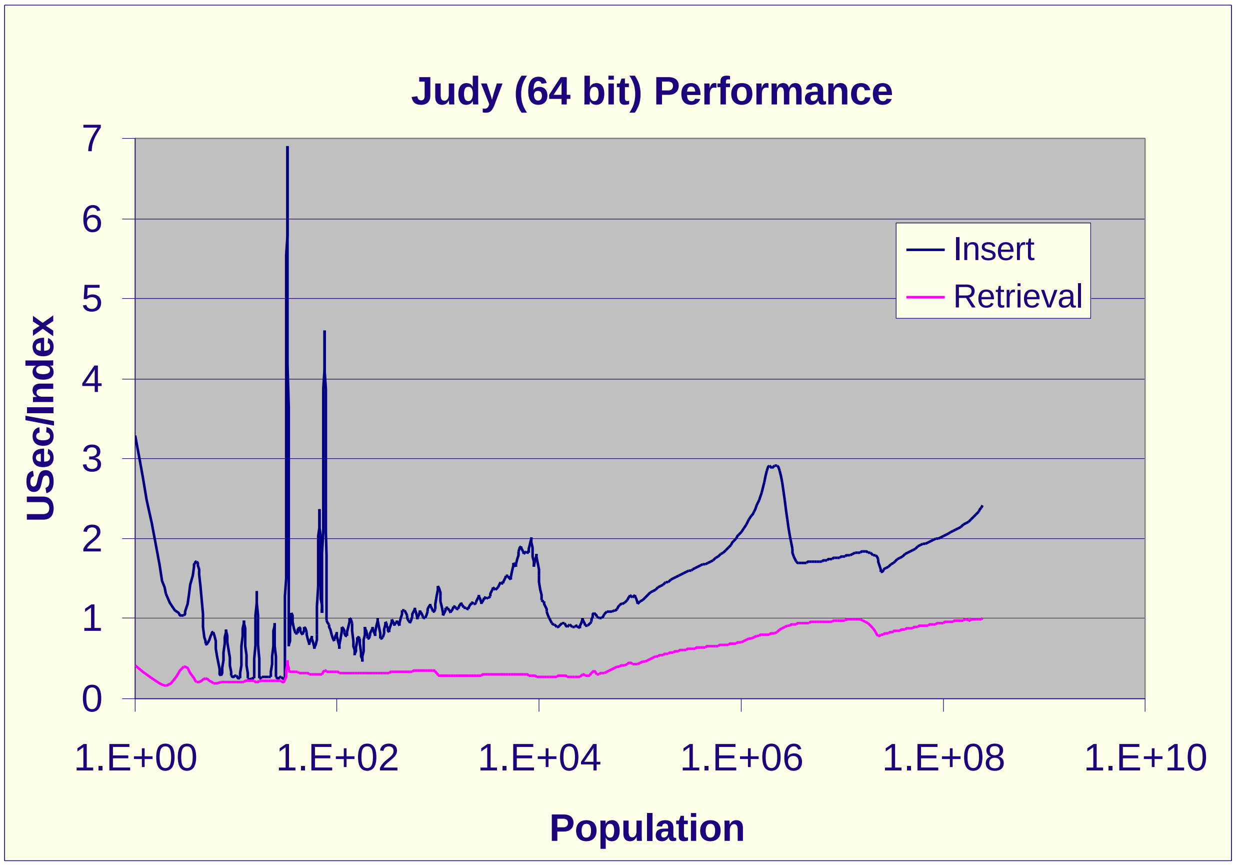
Development History

The Judy concept has been generating engineering interest at Hewlett-Packard for years. Early research and prototyping date back to 1981. The current Judy technology has been under active engineering development for four years. Judy has been proven in several internal HP tools and products, including a performance-profiling tool, a disk workload analyzer (WLA), and the OpenGL (Graphics Library).

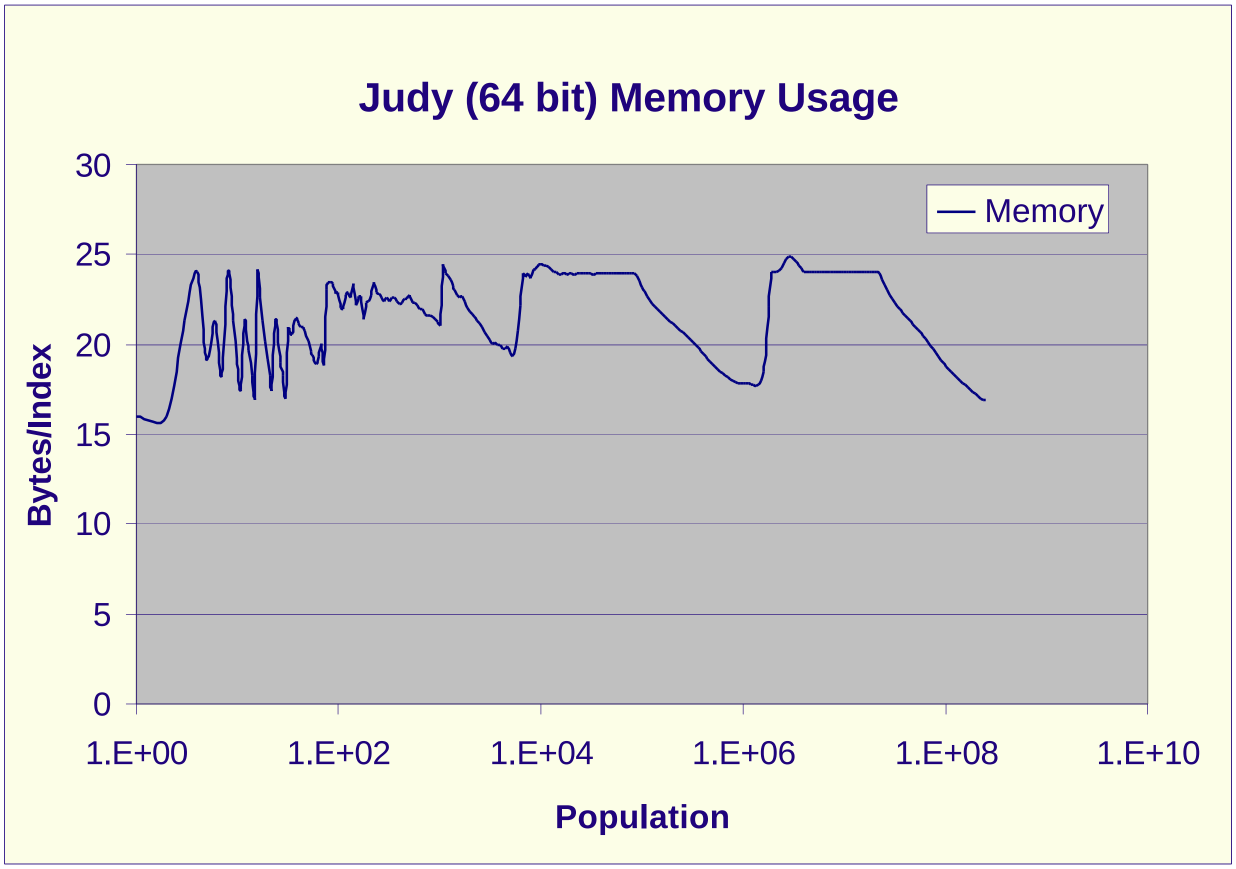
The Judy code has been tested and improved over several internally available iterations at Hewlett-Packard. This most recent version runs about twice as fast as the previous version. In this version, the JudyL functions were designed to maintain very good memory allocation of not more than 3 words per index. In fact, for large populations of clustered data, memory efficiency can be less than 2 words per index.

Performance and Memory

Judy performance is nearly linear [O($\log_{256}$N)] across very large populations. Judy is incredibly memory efficient…better than a linked list at small populations and better than most other structures at very large populations. The chart below shows the performance of 64-bit JudyL for insertion and retrieval of random data on an HP J6000 (with 8 Gbytes RAM). Random data (no clustering) represents Judy's worst case. Note that the y-axis is linear scale while the x-axis is log scale.



The chart below demonstrates the memory efficiency of 64-bit JudyL for random indexes, also the worst case for memory efficiency. Again, note that the y-axis is linear scale while the x-axis is log scale. For comparison, a doubly linked list would take 32 bytes per element; skip lists and binary trees take about 26 bytes per element, independent of the population.



Why is it Called Judy?

Many other "non-catchy" names have been used, such as AMLR (Associative Memory Lookup Routine) and SAMM (Sparse Array Memory Manager). In 1994, the inventor (Doug) realized his "relationship" with the algorithm paralleled the relationship with his sister – Judy (he fought with her growing up, but loves her now).

Doug with his sister Judy.

Coding Example - JudySL Sorting Example

You can find the code for this example and others on the Judy web site (after June 1, 2001): http://devresource.hp.com/judy/

```
// Judy demonstration code:  Judy equivalent of sort -u.  While Judy is not
// primarily intended as a sorting algorithm, this code demonstrates how
// you can use Judy to create a dynamic associative array.
//
// Usage:  <program> <file-to-sort>
//
// Code and comments are included if you want to modify the program to output
// duplicates as an exercise.
//
// Note:  If an input line is longer than BUFSIZ, it's broken into two or more
// lines.

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#include "Judy.h"

main (int argc, char ** argv)
{
    Pvoid_t  PJArray = (Pvoid_t) NULL;  // Judy array.
    PPvoid_t PPValue;                   // Judy array element.
    char     Index [BUFSIZ];            // string to sort.
    char *   Pc;                        // place in Index.
    FILE *   fpin;                      // to read file.
    JError_t JError;                    // Judy error structure.


// CHECK FOR REQUIRED INPUT FILE PARAMETER:

    if (argc != 2)
    {
        (void) fprintf (stderr, "Usage: %s <file-to-sort>\n", argv[0]);
        (void) fputs   ("Uses each line as a JudySL index.\n", stderr);
        exit (1);
    }


// OPEN INPUT FILE:

    if ((fpin = fopen (argv[1], "r")) == (FILE *) NULL)
    {
        (void) fprintf (stderr, "%s: Cannot open file \"%s\": %s "
                        "(errno = %d)\n", argv[0], argv[1], strerror (errno),
                        errno);
        exit (1);
    }

// INPUT/STORE LOOP:
//
// Read each input line (up to Index size) and save the line as an index into a
// JudySL array.  If the line doesn't overflow Index, it ends with a newline,
// which must be removed for sorting comparable to sort(1).
```

```
    while (fgets (Index, sizeof (Index), fpin) != (char *) NULL)
    {
        if ((Pc = strchr (Index, '\n')) != (char *) NULL)
            *Pc = '\0';                         // trim at newline.

        if ((PPValue = JudySLIns (& PJArray, Index, &JError))
            == PPJERR)
        {
            fprintf(stderr, "File: '%s', line: %d: Judy error: %d\n",
                    __FILE__, __LINE__, JU_ERRNO(&JError));
            exit (1);
        }

// To modify the program to output duplication counts, like uniq -d, or to emit
// multiple copies of repeated strings:

#ifdef notdef
        ++(*((ulong_t *) PPValue));
#endif
    }


// PRINT SORTED STRINGS:
//
// To output all strings and not just the unique ones, output Index multiple
// times = *((ulong_t *) PPValue).

    Index[0] = '\0';                            // start with smallest string.

    for ( PPValue  = JudySLFirst (PJArray, Index, &JError);
         (PPValue != (PPvoid_t) NULL) && (PPValue != PPJERR);
          PPValue  = JudySLNext  (PJArray, Index, &JError))
    {
        (void) puts (Index);                    // see comment above.
    }

// Error handling would go here.

    exit (0);
    /*NOTREACHED*/

} // main()
```

For More Information

For more information on HP's Judy Technology, visit the HP Judy Web site (after June 1, 2001):
`http://devresource.hp.com/judy/`

Legal Notices