Using B-trees to Implement Water: a Portable, High Performance, High-Level Language

A. Jaffer, M. Plusch, and R. Nilsson
Clear Methods, Inc.
One Broadway, 14th Floor
Cambridge, Massachusetts 02142 USA

Abstract - **To achieve high performance, the next generation of high-level programming languages should incorporate databases as core technology. Presented here are the design considerations for the Water language leading to the use of a (B-tree) Indexed Sequential Access Method database at its core.**

INTRODUCTION

As increasing CPU speed outpaces growth in memory bandwidth, programming languages must deal with aggregations of data in order to keep pace both with execution speed and programmer time. To scale with CPU speed, lower level language constructs require more and more sophistication from the (high-level) compiler. Higher level language constructs need only be optimized by the implementation language - provided they are the right constructs.

An example of this is the ***for_each*** construct of the Water language. This language's sole iteration construct maps over elements of vectors, maps over database keys and values, or maps over integers, and can collect results in a vector, in a database, or reduce them with a given function. The input keys can be arbitrarily filtered.

This is an example using ***for_each*** to iterate over a vector containing two strings.

```
<vector "zero" "one"/>.<for_each combiner=join>
  <join key "=" value "; "/>
</>
```

Returns: `"0=zero; 1=one; "`

The implementation of ***for_each*** has code optimized for the possible combinations of input type, map function, and filter. It also has code optimizing the common cases where primitive methods are given for filter, map, or reduce. Because ***for_each*** usually iterates multiple times per invocation, the time to dispatch to the correct code is negligible when amortized over the iterations.

The Water language had it origins in 1998 when Christopher Fry and Mike Plusch, the founders of Clear Methods, recognized both the potential and the limitations of XML and Web services. Water has since become a platform enabling businesses to make use of Web services and XML without the inherent limitations and complexity of traditional Web services development.

The first version of Water was released in 2001 to run on Java virtual machines. This first implementation suffers from slow operation and long startup times. In late 2006, a new higher-performance implementation was needed in order to achieve the performance level appropriate for a lightweight browser plug-in. In addition to achieving high performance, it was critical that the language be compatible with multiple operating systems and platforms.

The Water language is object-oriented. Object-key pairs are associated with methods and other values. Water has lexically-scoped environments; environment-variable pairs are associated with values. Water is reflexive; in the Java implementation, code is stored as objects. The top-level environment and root of the class hierarchy can grow to have a large number of associations.

The Java implementation of Water spends considerable time loading library code into the runtime image. Startup would be much faster if binary code objects could be saved and restored from a file.

Embedded platforms do not all support virtual memory. And many platforms (embedded or not) perform poorly as memory use by applications or plugins grows. To improve performance it is important to control RAM use by applications and plugins.

So we are looking for a core technology that:

- stores associations (in databases)
- has fast access times for both large and small databases
- can be saved to and restored from binary files
- has a bounded RAM footprint

B-trees [1] [2] [3] [4] have all of these properties. Such use of B-trees is not without precedent; created in 1966, the MUMPS (Massachusetts General Hospital Utility Multi-Programming System) and its derivatives are based on an Indexed Sequential Access Method (ISAM) database, most often B-trees.

We have adapted the ***WB*** B-tree library [5] for Water's use. It has the additional benefit of being thread-safe; critical update operations are protected by distributed locks; inter-thread communication is supported through mutual-exclusion operations provided in the application programmer interface. Thus ***WB*** can be used to support multiple sessions in a server or in a browser's multiple frames.

MULTILINGUAL PROGRAMMING

WB is written in a subset of the Algorithmic Language Scheme which can be translated to C by the ***Schlep*** compiler

[6] which is included in the WB distribution. At Clear Methods, Aubrey Jaffer and Ravi kiran Gorrepati adapted ***Schelp*** to create ***scm2java*** and ***scm2cs***, producing completely compatible implementations of ***WB*** in Java-1.5 and C#. This same translation technology is used for translating the Scheme sources for the Water compiler and runtime engine into C, Java, and C#.

The use of these translators means that compiler and engine development (and releases) can proceed in parallel with Water code development using any of Water's compatible platforms. The Scheme implementation (***SCM*** [7]) used for development does comprehensive bounds and type checking, eliminating the need for writing many program-logic checks into the source code of the Water compiler and engine.

Another mechanical translation is done by a simple bespoke Scheme program processing the data-representation design document, extracting the version, byte opcode, and (numerical) type assignments and producing source files which are included or otherwise used in the builds and runtime.

Java and C# provide garbage-collection. In C, the new Water implementation uses the Boehm conservative garbage collector [8] for temporary allocations.

Primitive Types

The keys and their associated values in ***WB*** are from 0 to 255 bytes in length. The 250 bytes are more than enough to host all the codepoints, identifiers, and numbers including bignums that users (other than number-theorists) need. Integers are from 1 byte to 249 bytes in length and are stored big-endian with a length prefix so they sort correctly as keys in B-trees. Water also encodes strings and binary objects smaller than 253 bytes as ***immediate*** objects.

Although there are techniques for extending B-tree keys and values in length, at some point it becomes burdensome for the runtime infrastructure to allocate and store large primitive types in the runtime image; doing so also can exceed the bounded RAM footprint. So the new Water implementation picks as its boundary 253 bytes. A string or binary object larger than this is given a unique identifier and its data is stored under numerical keys appended to its identifier. Whether a string or binary object is represented as a single immediate or as associations in a B-tree is not discernable to the user.

The index used for each string chunk is the UTF-8 codepoint offset of the end of the chunk from the beginning of the first chunk. Strings thus have $O(\log N)$ access time even though their UTF-8 representation has variable numbers of bytes per codepoint.

Object Encoding

The straightforward embedding of Water object structures into B-trees is that every record instance (classes are also record instances) has a unique identifier; and every slot corresponds to an association of the slot-value with that identifier combined with the slot-name. A slot-name is a non-negative integer or immediate string. A slot-value is either an immediate datum or an identifier for a (long string or) record.

A straightforward embedding of Water program expressions into B-trees builds on the object representation. Each expression is represented as a record. The ***_subject*** field (the object that gets the method call), if present, contains the literal datum or the identifier of the subject expression. The ***_method*** field contains the method-name string or the identifier of the method expression. Named arguments associate their keys (appended to the expression identifier) with their values or value expressions. Unnamed arguments associate their numerical order (0 ...) with their literal values or value expressions.

Variables and certain other strings used as keys or symbols are assigned unique identifiers; the forward and backward identification with strings being stored in B-trees. These identifiers are one to five bytes in length.

Although independent from other representation decisions, lexical bindings are also convenient to store in B-trees. Each environment has an identifier; and each variable (combined with the environment-identifier) is associated with its value. An environment's associations are deleted just before the environment is reused.

To support good error reporting, it is desirable to link every program expression to its location in a source file. This can be done simply in a B-tree while presenting no bloat or overhead to the code itself. A dedicated B-tree associates the identifier of each expression with its filename and offsets.

In ***WB***, a B-tree ***handle*** caches the last block visited, bypassing would-be full traversals from the B-tree's root for nearby references. To take advantage of this, the Water implementation uses six ***WB*** handles: string-to-variable, variable-to-string, bindings, records, program, and program-annotations.

Security

The six ***WB*** handles, along with directories to which a session has access, are the set of capabilities passed to routines in the Water compiler and runtime engine. They cannot be accessed or modified from a Water program. They are not stored in B-trees. Pointed to only from the call stack, they provide a measure of protection and isolation from other threads, which have separate call stacks.

Execution

Modern CPUs execute dozens of instructions in the time it takes to fetch one cache-line from main memory. Few applications today tend to be CPU-bound; most are memory- or cache-bound. (CPU-bound programs tend to be overrepresented in benchmark suites.) For all their benefits, access to small datums through B-trees does incur significant overhead. But for a runtime interpreter, multiple fetches from the straightforward embedding of program expressions precede each data access. Thus, the most productive area to optimize for overall performance is to reduce the bandwidth of program B-tree fetches.

Toward this end, we would like to aggregate a program expression into single B-tree value. But ***WB*** values are limited in length. So the aggregate expression format should also be space efficient. And the format should provide for overflow by being able to indicate that an expression is continued in the next B-tree association's value (***WB*** supports ISAM).

The aggregate expression format is a byte encoding with several types of codes. All the primitive methods are assigned single byte codes, as are prefix-markers. Identifiers, of which there are eight types (including symbol, long-string, method, and expression), have byte codes, the bottom two bits of which indicate the number of bytes following: 1, 2, 3, or 4. (Those identifiers are then between two and five bytes in size.)

For expressions there are markers delimiting the boundary between keyed and unkeyed arguments and the end of arguments. For the method definition, there are 24 codes indicating whether the following parameter is keyed or unkeyed, evaluated or unevaluated, required or optional, whether a default expression follows, and whether a type specifier follows.

SYSTEM STATE

As described here, all the state of a Water session except for the call stack is contained in its B-trees. ***WB*** being disk-backed, those B-trees are stored in a file on disk or other mass storage device. The time to run the 230kB Water executable, resume a 285kB saved session, compile and execute a trivial program, and exit takes about 6ms (3ms user + 3ms system) on a 3.0GHz Pentium-4 running Fedora-7 Linux. This time doesn't increase no matter how large the saved session is because ***WB*** reads only the blocks it needs from disk.

The ability to save program and data together into a format that runs on all platforms opens intriguing possibilities. Database files can contain their report generators, accessible with one click. Documents can adjust their formatting to suit the platform they are opened on.

ABOUT THE WATER LANGUAGE

Water is a secure, dynamic object-oriented language and database. It is an all-purpose language (and meta-language) that runs Web applications off-line in the browser or server-side [9]. The language is compatible with .NET, Java, and C on Windows, Linux and Mac OS X systems. Water handles all aspects of software including UI, logic, and database.

Water programs can store persistent data locally with Water's embedded object/XML database. The small footprint (<500kB) and instant-startup are well suited for running programs in the browser. HTML, CSS and JavaScript are naturally part of Water programs. The same Water program can be flexibly deployed to run either locally in the browser or on the server. Programs install automatically with one click.

The simplest ***Hello World*** program in Water is:

```
       "Hello Water!"
```

It displays the text *Hello Water!* in a browser window.

The following Water program uses a model-view-controller (MVC) design pattern.
(http://waterlanguage.org/examples/model_view_controller.h2o)

```
<class model_view_controller
       model_data=<v "sample string"/> />

<method model_view_controller.htm>
   <form action=<w .<controller_method/> />
    .model_data.<for_each combiner=insert>
      <div value/>
    </>
    <input name="an_input"
           value=.model_data.<last/> />
    <input type="submit" value="Submit"/>
   />
</>

<method model_view_controller.controller_method
        an_input=req>
   .model_data.<insert an_input/>
   _subject
</>

model_view_controller
```

Opening the URL runs the Water program in a browser and displays the screen shown in Fig 1.

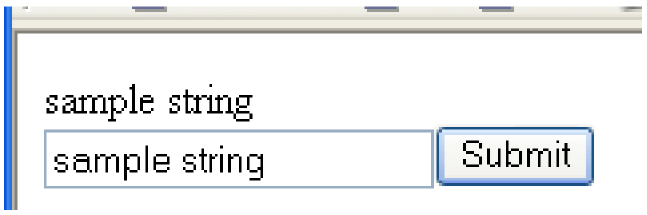

Fig. 1 Screen shot 1

This display of the application was created from a view implemented with the htm method. The model data is displayed in div tags using:

```
.model_data.<for_each combiner=insert>
              <div value/>
            </>
```

The input box displaying the last value in model data is created by:

```
<input name="an_input" value=.model_data.<last/> />
```

The submit button is created by:

```
<input type="submit" value="Submit"/>
```

If the user replaces ***sample string*** with ***Water*** in the input field, the program displays the screen shown in Fig 2.

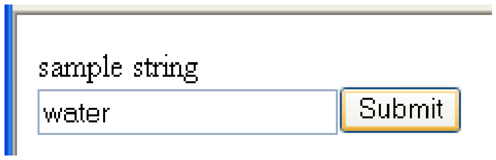

Fig. 2 Screen shot 2

Clicking ***Submit*** will call the controller_method and pass in the argument an_input with value ***Water***.

```
model_view_controller.<controller_method
                        an_input="Water"/>
```

When the controller method is called, it inserts an_input argument into the model data:

```
<method model_view_controller.controller_method
        an_input=req>
   .model_data.<insert an_input/>
   _subject
</>
```

This causes the application's presentation to refresh showing the value added to model data as in Fig. 3.

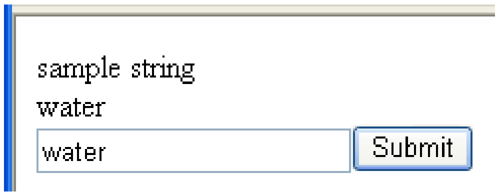

Fig. 3 Screen shot 3

REFERENCES

[1] Michael Ley, *B-Tree, Computer Science Bibliography* [Online], dblp.uni-trier.de, Universität Trier, Available: http://www.informatik.uni-trier.de/~ley/db/access/btree.html

[2] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," Acta Informatica, 1:173-189, 1972.

[3] Yehoshua Sagiv, "Concurrent Operations on B*-trees with Overtaking," JCSS 33(2): 275-296 (1986).

[4] W.E. Weihl and P. Wang, "Multi-version memory: Software cache management for concurrent B-trees," in *Proc. 2nd IEEE Symp. Parallel and Distributed Processing*, 1990, pp 650-655.

[5] R. Zito-Wolf, J. Finger, and A. Jaffer, *WB B-tree Library Reference Manual (2a2)* [Online], February 2008. Available: http://people.csail.mit.edu/jaffer/WB

[6] A. Jaffer, *Schlep: Scheme to C translator for a subset of Scheme* [Online], Available: http://people.csail.mit.edu/jaffer/Docupage/schlep.html

[7] A. Jaffer, *SCM Scheme Implementation Reference Manual (5e5*) [Online], February 2008, Available: http://people.csail.mit.edu/jaffer/SCM

[8] H. Boehm, A. Demers, and M. Weiser, *A garbage collector for C and C++* [Online], Available: http://www.hpl.hp.com/personal/Hans_Boehm/gc

[9] Plusch, Mike, *Water: Simplified Web Services and XML Programming* [Online], Available: http://waterlanguage.org/water_book_2002/index.htm