Code Specialization for Memory Efficient Hash Tries

Michael Steindorfer, Jurgen Vinju
Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
• memory usage vs runtime
• size of source code or binary
• platform specifics
Hash Tries
Hash Tries

Fast Immutable Data Structures on the JVM
Hash Tries

(Wide) Hash-Prefix Trees with Array Nodes
\{32, 2, 4098, 34\}
In contrast, Figure 1 illustrates a hash array mapped trie, while Figure 3 shows an array-based hash-table. By comparing the visualizations of both, we can identify the following list of disadvantages of HAMTs over array-based hash data structures:

- We first separate the hash codes in chunks of 5-bit to notate chunks as decimal values with ranges from 0 to 31. Then insertion places the values in a 32-nary tree (where each is encoded as a sparse array), based on the hash code prefixes. The tree structure gets expanded until every prefix can be unambiguously stored.

- To continue our example: 32 is inserted at the root node; 2 as well (because they do not share a common prefix). 4098 shares the prefix path unambiguously on level 2 from both.

- Note that a chunk size of 5-bit for 32-bit hash codes results in trees with a maximal depth of log_2(32) = 5.

- Insertion calculates the number's hash code: 7 (32), 2 (4098) = 16 (2). 4098 shares a common prefix with 32 (4098) = 16 (2).

- Lookup on level 3. 32 shares the prefix path unambiguously.

- Memory indirections and indexing into sparse arrays is less efficient than performing a single index-based lookup in continuous array.

- Provides evidence for our claims:

- ...
abstract class TrieSet implements java.util.Set {

    TrieNode root;
    int size;

    class TrieNode {
        int bitmap;
        Object[] contentAndSubTries;
    }
}
abstract class TrieSet
    implements java.util.Set {

    TrieNode root;
    int size;

    class TrieNode {
        int bitmap;
        Object[] contentAndSubTries;
    }
}
abstract class TrieSet
    implements java.util.Set {

    TrieNode root;
    int size;

    class TrieNode {
        int bitmap;
        Object[] contentAndSubTries;
    }
}
class TrieNode {
  int bitmap;
  Object[] contentAndSubTries;
}
class TrieNode {
    int bitmap;
    Object[] contentAndSubTries;
}

class NodeNode extends TrieNode {
    int bitmap;
    TrieNode nodeAtIndex0;
    TrieNode nodeAtIndex1;
}

class ElementNode extends TrieNode {
    int bitmap;
    Object keyAtIndex0;
    TrieNode nodeAtIndex1;
}

class NodeElement extends TrieNode {
    int bitmap;
    TrieNode nodeAtIndex0;
    Object keyAtIndex1;
}

...
Exponential
Number of Specializations
<table>
<thead>
<tr>
<th>Arities</th>
<th>% of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>≤4</td>
<td>82%</td>
</tr>
<tr>
<td>≤8</td>
<td>86%</td>
</tr>
<tr>
<td>≤12</td>
<td>90%</td>
</tr>
<tr>
<td>Arities</td>
<td>Specializations</td>
</tr>
<tr>
<td>---------</td>
<td>----------------</td>
</tr>
<tr>
<td>$\leq 4$</td>
<td>31</td>
</tr>
<tr>
<td>$\leq 8$</td>
<td>511</td>
</tr>
<tr>
<td>$\leq 12$</td>
<td>8191</td>
</tr>
</tbody>
</table>
Avoiding Permutations
<table>
<thead>
<tr>
<th>Arities</th>
<th>Specializations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\leq 4$</td>
<td>15 (31)</td>
</tr>
<tr>
<td>$\leq 8$</td>
<td>45 (511)</td>
</tr>
<tr>
<td>$\leq 12$</td>
<td>91 (8191)</td>
</tr>
</tbody>
</table>
abstract class TrieSet implements java.util.Set {
    TrieNode root;  int size;

    interface TrieNode { ... } 
    ...

class NodeNode extends TrieNode {
    int bitmap;
    TrieNode nodeAtIndex0;
    TrieNode nodeAtIndex1;
}
class ElementNode extends TrieNode {
    int bitmap;
    Object keyAtIndex0;
    TrieNode nodeAtIndex1;
}
class NodeElement extends TrieNode {
    int bitmap;
    TrieNode nodeAtIndex0;
    Object keyAtIndex1;
}
class ElementElement extends TrieNode {
    int bitmap;
    Object keyAtIndex0;
    Object keyAtIndex1;
}
abstract class TrieSet implements java.util.Set {
    TrieNode root;  int size;

    interface TrieNode { ... } ...

class NodeNode extends TrieNode {
    byte pos1; TrieNode nodeAtPos1;
    byte pos2; TrieNode nodeAtPos2;
}
class ElementNode extends TrieNode {
    byte pos1; Object keyAtPos1;
    byte pos2; TrieNode nodeAtPos2;
}
class NodeElement extends TrieNode {
    byte pos1; TrieNode nodeAtPos1;
    byte pos2; Object keyAtPos2;
}
class ElementElement extends TrieNode {
    byte pos1; Object keyAtPos1;
    byte pos2; Object keyAtPos2;
}
...
Lookup Performance (lower is better)

Map

Generic | Specialized 0-4 | Specialized 0-8 | Specialized 0-12

100% | 130% | 138% | 138%
Memory Usage (lower is better)

- Map:
  - Generic: 100%
  - 0-4: 62%
  - 0-8: 46%
  - 0-12: 45%

- Set:
  - Generic: 100%
  - 0-4: 52%
  - 0-8: 23%
  - 0-12: 22%
Memory Footprint Compared To Competition (lower is better)

- Specialized 0-8: 1x for Map, 1x for Set
- Clojure: 1,6x for Map, 2,2x for Set
- Scala: 4,9x for Map, 3,75x for Set
worst hash distribution

->

good memory performance
best hash distribution -> worst memory performance
best hash distribution -> worst memory performance