STRUCTURES DE DONNEES EXOTIQUES

17h - 17h50  - Salle Seine A
• Sam BESSALAH

• Independent Software Developer,

• Works for startups, finance shops, mostly in Big Data, Machine Learning related projects.

• Rambling on twitter as @samklr
Why care about data structures?
Powerful libraries, powerful frameworks. But ..

« If all you have is a hammer, everything looks like a nail ».  

Abraham H. Maslow
SKIPLISTS
Efficient structure for sorted data sets

Time complexity for basic operations:
Insertion in $O(\log N)$
Removal in $O(\log N)$
Contains and Retrieval in $O(\log N)$
Range operations in $O(\log N)$
Find the $k$-th element in the set in $O(\log N)$
Start with a simple linked list
Add extra levels for express lines
Search for a value looks like this
Insert \((X)\)

Search \(X\) to find its place in the bottom list.

\((Remember\ the\ bottom\ list\ contains\ all\ the\ elements\ ).\)

Find which other list should contain \(X\)

Use a controlled probabilistic distribution.

Flip a coin;

if HEADS

  Promote \(x\) to next level up, then flip again

At the end we end up with a distribution of the data like this

- \(\frac{1}{2}\) of the elements promoted 0 level
- \(\frac{1}{4}\) of the elements promoted 1 level
- \(1/8\) of the elements promoted 2 levels
- and so on
Remove \( (X) \)
Find \( X \) in all the levels it participates and delete
If One or more of the upper levels are empty, remove them.
void insert(E value) {
    SkipNode<E> x = header;
    SkipNode<E>[] update = new SkipNode[MAX_LEVEL + 1];

    for (int i = level; i >= 0; i--) {
        while (x.forward[i] != null && x.forward[i].value.compareTo(value) < 0) {
            x = x.forward[i];
        }
        update[i] = x;
    }
    x = x.forward[0];

    if (x == null || !x.value.equals(value)) {
        int lvl = randomLevel();
        if (lvl > level) {
            for (int i = level + 1; i <= lvl; i++) {
                update[i] = header;
            }
            level = lvl;
        }
    }
}
x = new SkipNode<E>(lvl, value);
    for (int i = 0; i <= lvl; i++) {
        x.forward[i] = update[i].forward[i];
        update[i].forward[i] = x;
    }
}

private int getLevel(){  // Coin Flipping
    int lvl = (int)(Math.log(1.-Math.random())/Math.log(1.-P));
    return Math.min(lvl, MAX_LEVEL);
}
Fast structure on average, very fast in practice

Can be implemented in a thread safe way without locking the entire structure, and instead acting on pointers. In a lock free fashion, by using CAS instructions.

In Java since JDK 1.6 within the collections `java.util.concurrent.ConcurrentSkipListMap` and `java.util.concurrent.ConcurrentSkipListSet`. Both are non blocking, thread safe data structures with locality of reference properties. Ideal for cache implementation.
CAS instruction

CAS(address, expected_value, new_value)

Atomically replaces the value at the address with the new_value if it is equal to the expected_value.

In one instruction CMPXCHG

Returns true if successful, false otherwise.
Drawbacks

They can be memory hungry, by being space inefficient.
TRIES
- Ordered Tree data structure, used to store associative arrays. Usually, encoded keys through traversal, in the nodes of the tree with value in the leaves.

- Used for dictionary (Map), word completion, web requests parsing, etc.

- Time complexity in $O(k)$ where $k$ is the length of searched string. Where it usually is $O$(length of the tree)
HASH ARRAY MAPPED TRIE
(HAMT)

Functional data structure, for fast association map

Based on the idea of hashing a key, and storing the value on a trie.

Each node is an array of size 32, pointing to children nodes of same size with maximum Seven deferencement, for fast key look ups.
How does it work?

During association of $K \rightarrow V$, compute hashes yielding an Integer or a long, coded on the JVM in 32 bits.

Partition those bits to blocks of 5 bits, representing a level in the tree structure, from the right most (root) to the left most (leaves).

The colored nodes have between 2 and 31 children.

Each colored node maintain a bitmap, encoding how many children the nodes contains, and their indexes in the array.
Trie Structure (from Rich Hikey)
Hash Array Mapped Tries (HAMT)
Hash Array Mapped Tries (HAMT)
Hash Array Mapped Tries (HAMT)
Hash Array Mapped Tries (HAMT)

$16 = 010000_2$
Hash Array Mapped Tries (HAMT)
Hash Array Mapped Tries (HAMT)

4 = 000100₂

0 16
Hash Array Mapped Tries (HAMT)

$4 = 000100_2$

Diagram showing the binary representation of 4 and how it corresponds to the trie structure with the number 16 at a specific node.
Hash Array Mapped Tries (HAMT)
Hash Array Mapped Tries (HAMT)

12 = 001100₂
Hash Array Mapped Tries (HAMT)

12 = 001100₂
Hash Array Mapped Tries (HAMT)
Hash Array Mapped Tries (HAMT)
Hash Array Mapped Tries (HAMT)
Hash Array Mapped Tries (HAMT)
Hash Array Mapped Tries (HAMT)
Hash Array Mapped Tries (HAMT)
Hash Array Mapped Tries (HAMT)

Too much space!
Hash Array Mapped Tries (HAMT)
Hash Array Mapped Tries (HAMT)

Linear search at every level - slow!
Hash Array Mapped Tries (HAMT)

Solution – bitmap index!
Relying on BITPOP instruction.
Hash Array Mapped Tries (HAMT)

BITPOP(((1 << ((hc >> lev) & 1F)) - 1) & BMP)
Complexity almost all in log32 N \textasciitilde O(7) \textasciitilde O(1) 

Requires only 7 max, array deferencements.

<table>
<thead>
<tr>
<th>O(1)</th>
<th>O(log n)</th>
<th>O(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Append</td>
<td>First Insert Last</td>
<td>concat insert prepend</td>
</tr>
<tr>
<td>K-th Update</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hash Array Mapped Tries (HAMT)

• advantages:
  • low space consumption and shrinking
  • no contiguous memory region required
  • fast – logarithmic complexity, but with a low constant factor
  • used as efficient immutable maps

Clojure's PersistentHashMap and PersistentVector, Scala's Mutable Map.

• no global resize phase – real time applications, potentially more scalable concurrent operations?
Concurrent Trie (Ctrie)

- goals:
  - thread-safe concurrent trie
  - maintain the advantages of HAMT
  - rely solely on CAS instructions
  - ensure lock-freedom and linearizability

- lookup – probably same as for HAMT
Lock Free Concurrent Trie (C Trie)
SKETCHES
(Or how to remove the fat in your datasets)
BLOOM FILTERS
Probabilistic data structures, designed to tell rapidly in a memory efficient way, whether an element is absent or not from a set.

Made of Array of bits B of length n, and a hash function h

**Insert**(X): for all i in set, B[h(x,i)] = 1
**Query**(X): return FALSE if for all i, B[h(x,i)] = 0

Only returns if all k bits are set.
We can pick the error rate and optimize the size of the filter to match requirements.

\[ \text{False Positive rate} \approx \left( 1 - \left(1 - \frac{1}{m}\right) \right)^k \approx \left( 1 - e^{-\frac{kn}{m}} \right)^k \]

Optimal # of hash functions \( k \) \approx \ln 2 \frac{m}{n} \approx 0.7 \frac{m}{n}

**10,000 words, 1% error rate:**
- \( m = 10,000 \times 10 \text{ bits} \approx 12 \text{ kb of memory} \)
- \( k = 0.7 \times \frac{m}{10,000} = 7 \text{ hash functions} \)

**10,000 words, 0.1% error rate:**
- \( m = 10,000 \times 15 \text{ bits} \approx 18 \text{ kb of memory} \)
- \( k = 0.7 \times \frac{m}{10,000} = 11 \text{ hash functions} \)
TRADE OFFS

More hashes enduce more accurate results, but render the sketch less space efficient.

Choice of the hash function is important.

Cryptographic hashes are great; because evenly distributed, with less collisions.

Watch out to time spent computing your hashes.
Cool properties

Intersection and Union through AND and OR bitwise operations

No False negatives

For union, helps with parallel construction of BF

Fast approximation of set union, by using bit map instead of set manipulation
Handling Deletion

Bloom filters can handle insertions, but not deletions.

If deleting $x_i$ means resetting 1s to 0s, then deleting $x_i$ will “delete” $x_j$.

Solution: Counting Bloom Filter
COUNTING BLOOM FILTERS

Keeps track of inserts

- Query(X) : return TRUE if for all i $b[h(x,i)] > 0$
- Insert(X) : if $(query(x) == false)$ { //don't insert twice
  For all i $b[h(x,i)]++$
}
- Remove(X) : if $(query(x) == true)$ { //don't remove absents
  For all i, $b[h(x,i)]--$
}
Usages

- Fast web events tracking
- IO optimisations in databases like Cassandra, Hadoop, Hbases ..
- Network IP filtering ...
Guava Bloom Filters

Default gives a 3% error. With a MurMurHashV3 function.

Creating the BloomFilter

```java
BloomFilter bloomFilter = BloomFilter.create(Funnels.byteArrayFunnel(), 1000);
```

//Putting elements into the filter

//A BigInteger representing a key of some sort

```java
bloomFilter.put(bigInteger.toByteArray());
```

//Testing for element in set

```java
boolean maybeContained = bloomFilter.mayContain(bitIntegerII.toByteArray());
```
// With a custom filter

class BigIntegerFunnel implements Funnel<BigInteger> {
    @Override
    public void funnel(BigInteger from, Sink into) {
        into.putBytes(from.toByteArray());
    }
}

// Creating the BloomFilter
BloomFilter bloomFilter = BloomFilter.create(new BigIntegerFunnel(), 1000);

// Putting elements into the filter
bloomFilter.put(bigInteger);

// Testing for element in set
boolean mayBeContained = bloomFilter.mayContain(bitIntegerII);
COUNT MIN SKETCHES
Family of memory efficient data structures, that can estimate frequency-related properties of the data set.

Used to find occurrences of an element in a set, in time / space efficient way, with a tunable accuracy.

E.g Find heavy hitters elements; perform range queries (where the goal is to find the sum of frequencies of Elements in a certain range ), estimate quantiles.
How does it work?
Two-dimensional array (d x w) of integer counters. When a value arrives, it is mapped to one position at each of d rows using d different and preferably independent hash functions.
d : number of hash functions
w : hash array size
Algorithm:

insert(x):

for i = 1 to d

M[i, h(x, i)] ++  // Like counting bloom filters

query(x):

return min {h(x, i) for all 1 ≤ i ≤ d}
Accuracy depends on the ratio between sketch size and number of expected data. Works better with highly uncorrelated, unstructured data.

For highly skewed data, use noise estimation, and compute the median estimation.
HEAVY HITTERS
*(through count min sketches)*

Find all the elements in a data sets with frequencies over a Fixed threshold. K percent of the total number in the set.

Use a count min sketched algorithm.

Use case: detect most traffic consuming IP addresses, thwart DDoS attacks by blacklisting those ips. Detect market prices with highest bids swings.
**Algorithm.**

1. Maintain a Count-min sketch of all the element in the set.
2. Maintain a heap of top elements, initially empty, and a Counter N, of already processed elements.
3. For each element in the set
   - Add it in the set
   - Estimate the Frequency of the element. If higher than the threshold k*N, add it to heap. Continuously clean the heap of all the elements below the new threshold.
OTHER INTERESTING SKETCHES
10^7 elements
10^6 distinct values
domain of 32-bit integers

40 MB
Raw Data

0.6 MB
Membership Query
with 4% error – Bloom Filter

4 MB
Exact Membership Query,
Cardinality Estimation – Sorted IDs or Hash Table

48 KB
Frequencies of top-100 most frequent elements with 4% error – Count-Min Sketch

14 KB
Top-100 most frequent elements with 4% error – Stream-Summary

7 MB
10^6 pairs
{ 32-bit value,
24-bit counter }

2 KB
Cardinality Estimation
with 4% error – Loglog Counter

125 KB
Cardinality Estimation
with 4% error – Linear Counter

Exact Frequency
Estimation, Range Query – Sorted Table or Hash Map

http://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/
**Bibliographies & Blogs.**


Libraries.

http://github.com/slearspring/stream-lib

org.apache.cassandra.utils.{MurmurHashV3, BloomFilter}

Google Guava.

Ideal Hash Trees by Phil BagWell


Concurrent Tries in the Scala Parallel Collections

SkipLists By William Pugh.