High Performance Data Structures: Theory and Practice

Roman Elizarov
Devexperts, 2006

e1

High performance?

NO!!! We'll talk about different "High Performance".
One that aimed at avoiding all those racks and racks….

Slide 2

e1 elizarov; 17.11.2006

Programs = Algorithms + Data Structures

- This classical quote (N. Wirth) sounds funny nowadays.
  - You rarely hear algorithms and data structures discussed in modern business-oriented software development.
  - Why?

Classical algorithms analysis

- As taught in classical books:
  - Knuth, The Art of Programming
  - Wirth, Algorithms & Data Structures
  - Aho & Ullman, Data Structures & Algorithms
  - Cormen et al, Introduction to Algorithms
- Algorithms and data structures are analyzed based on their asymptotical performance for N elements or operations – O(N), O(N log N), $O(N^2)$, $O(N^3)$, etc.
  - "Effective" algorithms are of the most interest

Performance matters…

- … for large N values

N	N log N	N^2	N^3
10	20	100	1K
100	300	10K	1M
1K	4K	1M	1G
10K	50K	100M	1T

Performance… anybody?

- Most business systems have 3-tier architectures: data, logic, and presentation.
  - Data layer is usually implemented in DBMS. That’s where a bulk of data is located (N > 10K).
  - Logic layer works only with small query results from database (N ~ 100).
  - Presentation layer similarly processes small human-consumable portions of data (N ~ 100).

Does it matter for small Ns?

- Modern entry-level systems perform “just” ~1Gops/s
- How many ops/s we could make (in red)?

N	N log N	N^2	N^3
10, 100M	20, 50M	100, 10M	1K, 1M
100, 10M	300, 3M	10K, 100K	1M, 1K
1K, 1M	4K, 250K	1M, 1K	1G, 1
10K, 100K	50K, 20K	100M, 10	1T, 0.001

Business … as usual

- Usually only DBMS vendor’s developers are facing large Ns (work with considerable amounts of data and operations on them).
- Most application developer never face large sets of data in their entire career.
  - They don’t have to!
- What happens when those developers suddenly face it?
  - Disaster.

Performance-critical areas

- Processing of large databases (N > 10K)
  - Mostly solved problem by DBMS vendors, but may require special skills and understanding.
- Real-time processing of events:
  - Telemetry
  - Telecommunications
  - Real-time financial transactions
  - Real-time monitoring
  - Your examples here

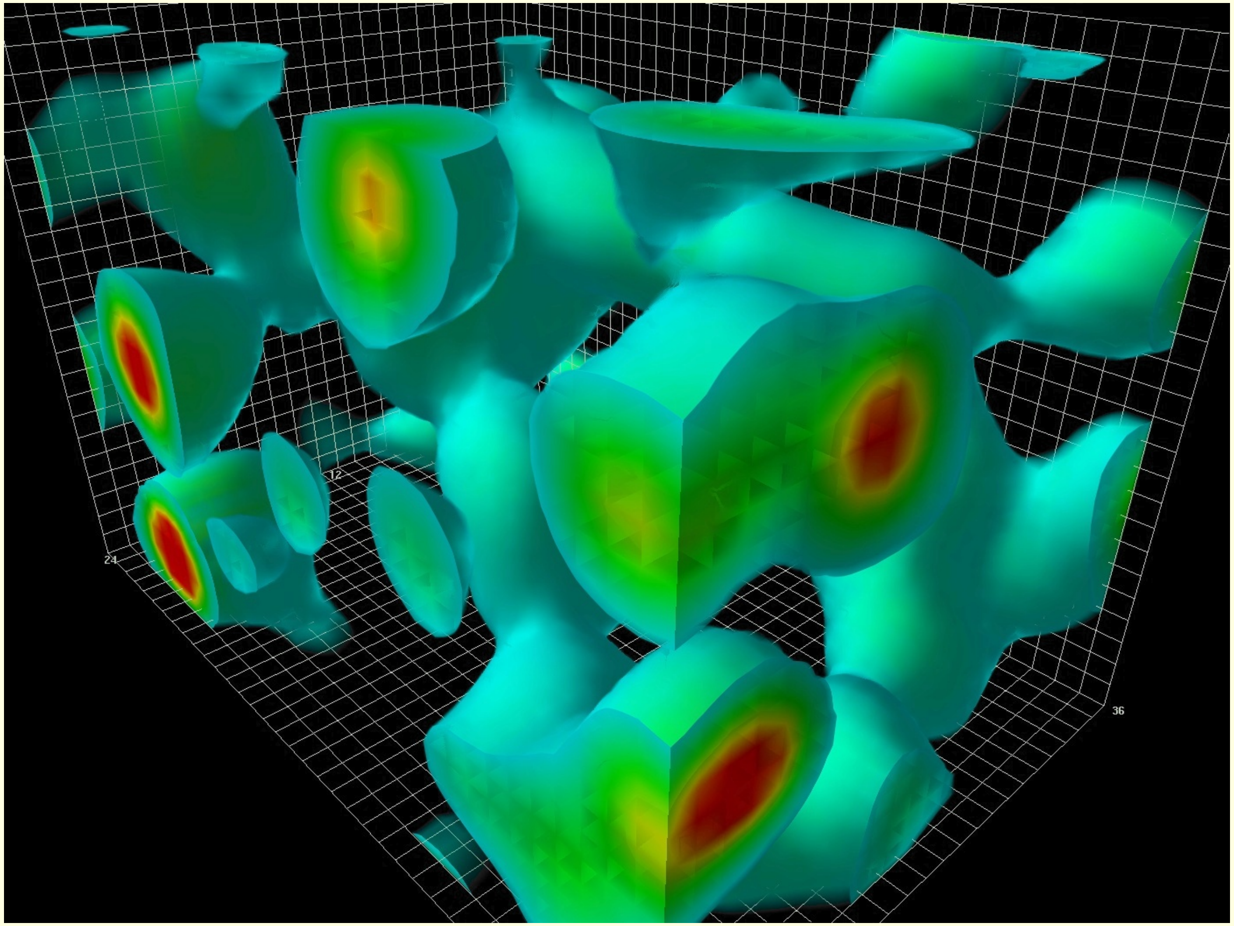
Not solved?

- DBMSs on a regular hardware perform around ~1-10K transactions per second at most.
  - Clearly not enough if you have > 100K quotes per second from all exchanges around the world to process.
- A lot of hand-coding is required when you try to receive, process, store, and/or forward huge amounts of data in real-time.
  - How would you even parse > 1Mbytes/s of incoming network traffic?

But all algos & DSs are there to use!

- All the modern languages (C++, Java, C#) have standard libraries with:
  - Array & linked lists, deques, stacks;
  - Priority queues;
  - Tree (sorted) maps & sets;
  - Hash maps & sets;
  - Sorting algorithms.
- All with the best theoretical performance
  - What else a sophisticated high performance software might ever need?

Theory



Practice vs Theory

- In practice, <u>if</u> performance matters you'd like to have every conceivable bit of it
  - You would not write in assembler (huh?) …
  - … but for some applications even this is not the last practical resort (out of topic, though)
- In theory it is just an <u>asymptotical</u> performance that matters.
  - How come it is not enough?

Reality strikes back

- Modern hardware has exceedingly complex design that affects software performance on many levels.
  - For business systems it usually boils down to <u>memory subsystem</u>.
  - Now, scientific software might also heavily depend on FP & command scheduling details (but that is out of topic for this discussion).
- Deep understanding of the modern hardware is required to get most of its potential.

A very simple demo

```
// Constants
int KB = 1024;
int MB = KB * KB;
int SIZE = 256 * MB;

// Data (randomly filled)
int[] data = new int[SIZE / 4];
int[] ofs = new int[SIZE / 4];
int res; // temporary var

// Sequential read of data
for (int i = 0; i < data.length; i++)
        res += data[i];

// Random read of data, sequential read of ofs
for (int i = 0; i < ofs.length; i++)
        res += data[ofs[i]];
```

… and results

Sequential read of data	140 ms
Random read of data, sequential ofs	2750 ms

* On 2GHz Intel Pentium M Processor

- It means that addition of random read in the second test slowed it down by 2610 ms.
  - Random read is ~ 18 times slower!

Modern computing

- Modern memory has very high latency compared to system clock speed.
  - But it has high throughput (if you can use it).
- Latency problems are partially addressed by cache hierarchy.
  - But it will not help you with really large data.
- Why is it designed that way?

Modern computing cont'd

- Modern computing hardware is mostly optimized for multimedia & streaming data processing.
  - Video, Audio, Pictures.
  - Encoding/decoding.
- All subsystems are oriented for those goals:
  - Special SIMD (vector) instruction sets;
  - Caches that read a range of memory at once;
  - Prefetch of next memory locations.
- But few business (server) applications really care about high-speed video encoding!

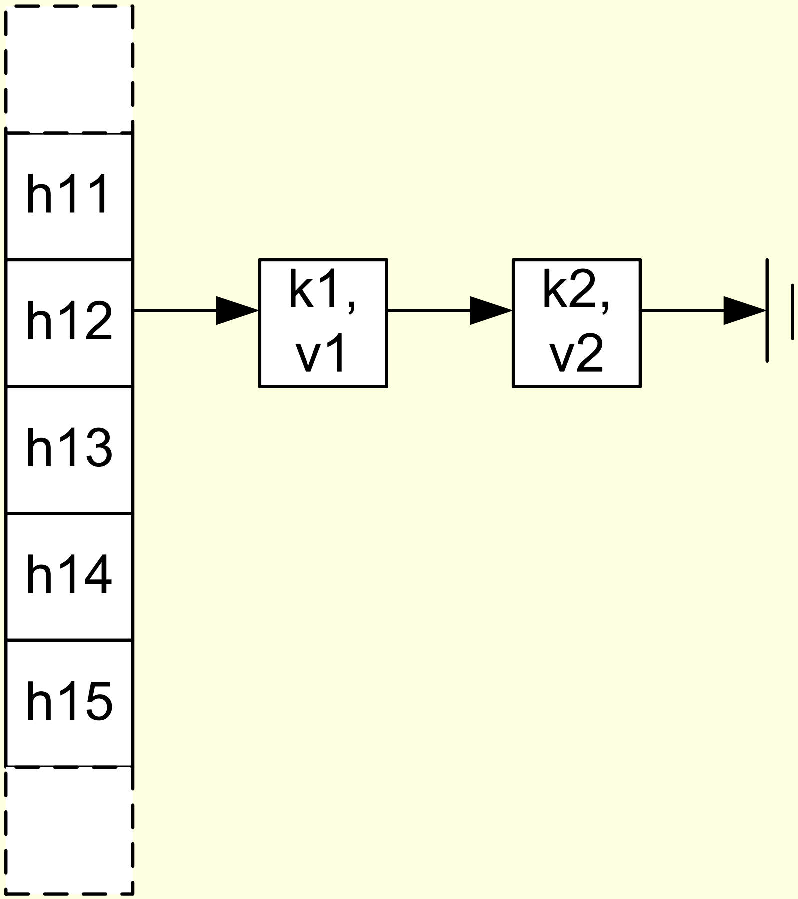
A problem (as example)

- What if we have > 100K event/s from 10-100K sources that we need to sort out by source and process separately?
  - Quotes, telemetry, etc.
  - It may all come via a single network stream.
- We would need to randomly use a large portion of memory to keep all information related to a single source.
  - We'll need a dictionary to find our source-related information.

Hash tables

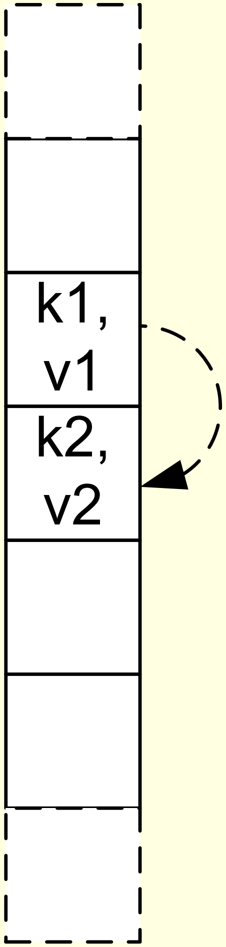
- Hash tables are usually the prime choice:
  - O(1) amortized update/access time.
  - They are available in all standard libraries.
  - But they are coded up to classical recipes.
- Knuth names several ways to resolve collisions in hash functions:
  - **Chaining (the most popular in practice)**
  - Open addressing (linear probing, quadratic probing, double hashing)

Chaining



- Even a successful hash lookup requires access to several memory locations.
  - Even when chains are of the shortest possible length (one)!

Linear addressing



- Cells are implicitly linked (next or previous one is checked on collision).
  - Typical cache would load all information in a single request
  - … even when a chain of liked cells is long.

Let’s check it out

```
// good things never work without magic
int MAGIC = 0xC96B5A35;

// data structure elements
Object[] a; // hash-table itself – 2*i – keys, 2*i+1 – values
int shift; // shift for hash-code

// Lookup algorithm. Object key is on input
int i = ((key.hashCode() * MAGIC) >>> shift) << 1;
Object k;
while (!key.equals(k = a[i])) {
        if (k == null)
                return null;
        if (i == 0)
                i = a.length;
        i -= 2;
}
return a[i + 1];
```

… and results

Chaining (code from a library)	1407 ms
Linear addressing (our code)	750 ms

* On 2GHz Intel Pentium M Processor

** Key hashes are from 0 to 3999999, values random

- Linear addressing is almost x2 as fast, even though:
  - We work with object keys, so some random memory access is required anyway (to follow a link to the key object for equality test).

Conclusions

- Number of memory “blocks” accessed is what actually matters a lot.
  - Typically, the fewer memory your data structure consumes the faster it is.
- In classical analysis of algorithms there is a class of algorithms for external memory that are analyzed not for their asymptotical performance, but for number of blocks of external memory they access.
  - That is what needed for modern hardware!

Emerging science

- There is an emerging class of “cache oblivious” algorithms that perform equally good (is some sense) on any memory hierarchy with any [unknown] cache sizes.
  - Practical and theoretical results are limited.
  - Lots of room for actual and new research.

Thank you for your attention

- Questions?

Roman Elizarov
Devexperts, 2006
elizarov@devexperts.com