Multi-part Nanocubes

Partitioning, Parallelization & Merging
New Memory Management
Windows and 32-bit support

Lukas Scharlau

Pyroluk@fh-muenster.de

BACHELOR THESIS

in computer science at the

FH MÜNSTER
University of Applied Sciences

Department of Electrical Engineering and Computer Science

Supervisors:
Prof. Dr. Gernot Bauer, FH Münster
Dr. David Rosenberg, Bloomberg LP, formerly YP Mobile Labs

Abstract

This thesis describes the development of Multi-part Nanocubes. It is a further development of *Nanocubes*, an in-memory data structure for spatiotemporal data cubes. Partitioning the structure to parallelize the build process as well as merging query results is the principal part of this document. Furthermore, a new memory management (slab allocation with offset pointers) was implemented to enable 32-bit support and faster load times of already built nanocubes. Porting the project to Windows and implementing on-the-fly compression and decompression of nanocube files is also described.

Contents

1 Declaration of Authorship 4

2 Acknowledgments 5

3 Introduction 6
- 3.1 Motivation . . . 6
- 3.2 Objective . . . 7

4 Nanocubes 9
- 4.1 Building a Nanocube from raw data . . . 10

5 Multi-part Nanocubes 13
- 5.1 Quadtree Partition . . . 13
- 5.2 C++11 Implementation . . . 17
  - 5.2.1 Multi-part command line options . . . 17
  - 5.2.2 Multithreading . . . 18
  - 5.2.3 Merging query results . . . 21
- 5.3 Benchmarks . . . 24
  - 5.3.1 Procedure . . . 25
  - 5.3.2 Results . . . 25
  - 5.3.3 Profiling: Even more Speed with TCMalloc . . . 37

6 Save and Load 44
- 6.1 Compression . . . 45
- 6.2 32-bit Support . . . 50

7 Nanocubes on Windows 51

8 Future 53

9 Appendix 54
- 9.1 Pseudocode: Merging two Nanocubes . . . 54
- 9.2 UML class diagram . . . 55

1 Declaration of Authorship

I hereby certify that this thesis has been composed by me and is based on my own work, unless stated otherwise. No other person's work has been used without due acknowledgment in this thesis. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged.

Steinfurt, September 9, 2016

2 Acknowledgments

I thank Prof. Dr. Gernot Bauer who made it possible for me to travel and experience New York. Moreover, I thank David Rosenberg and David Rosenstrauch for hiring me to work at Yellow Pages and work on interesting topics. I thank Peter Thai, Jason Uechi and David Rosenberg for introducing the Nanocube project, which later became the topic of my thesis. I especially thank David Rosenberg for helping me to understand this novel technology.

A huge thanks goes to Lauro Lins (AT&T)[1], the main developer of Nanocubes, who worked with me on how to realize Multi-part Nanocubes. His genius mind lead us to the simple and clever way of partitioning the data structure as described in this document. Using slab allocation and offset pointers was his idea, too. Kudos!

Lastly, I thank Parth Savani, Peter Thai, Timo Schwarte, Lauro Lins and David Rosenberg for proofreading my work.

[1]http://www.research.att.com/people/Lins_Lauro_D

3 Introduction

3.1 Motivation

Yellow Pages is known for their telephone directory of businesses. Nowadays, they also offer their services online and are active in the online marketing industry. Among other things they offer personalized location based advertising. For example, they can restrict ad campaigns to select U.S. states or only show the ad if you are close to a store of the advertiser. Many more filters can be applied to target a specific group of people.

Real-time bidding (RTB) is a modern way of online marketing adopted by the mobile and desktop advertising industry. Companies like Smaato Inc., MoPub Inc., Nexage LLC, Tapad Inc., and many more provide software development kits (SDK) that app/web developers use to take part in their RTB system. Smart phone apps and websites using the SDK will trigger an auction, offering their advertising space every time they are about to show an advertisement. The auction is hosted on the RTB ad exchange platform the SDK is bound to. Companies running Demand-side platforms (DSP) like Yellow Pages (YP) and others, have contracts with ad exchange platform providers to take part of the advertising space auctions. They receive a notification with information about each auction. If a DSP calculates (within milliseconds) that an advertising space is worth buying for their customer, they will respond with the amount of money they are willing to spend. If the DSP wins the bid, their client's advertisement will be shown.

The requests sent for an auction contain, among many other things, an ad exchange platform specific user id, operating system, app name and if procurable even the *exact position* of the phone. This allows DSPs to build user profiles over time and use them to make better decisions while bidding. With the help of statistics and by cross-referencing external data, those profiles can contain predictions of the phone owner's home and work address, income range and more.

Huge amounts of data are sent through RTB systems. Yellow Pages for instance collected over a petabyte of raw, uncompressed bid requests in text format from several ad exchange platforms in about 600 days. The requests answered during a single Papa John's Pizza advertising campaign with 16746730 impressions take up to 15 GB of disk space in raw text format.

Yellow Pages is interested in visualizing this kind of *big data*. Campaign managers and sales people could gain confidence about their advertising strategy and the data in general by filtering and visually browsing through a heat map generated from a data set of interest. Not only would this allow them to check, examine and verify a strategy, it would also help them to plan a sophisticated strategy by, for example, analyzing the movement of (specific) people over time to make user stories evident. Nobody else offers this kind of information.

Considering the magnitude of data, it became apparent that a real-time visualization of such *big data* would require computing power of the same order of magnitude. In other words: A big server was needed. Fortunately, this kind of problem can be reduced to the point that it can now be solved by a modern laptop. The award winning technology **Nanocubes**, developed at the Information Visualization department at AT&T Labs Research, is designed to serve real-time visualizations of such huge spatiotemporal[1] data sets. The technology aims to minimize memory consumption and maximize query speed. Furthermore, the interface and the level of detail of the visualization is market leading, which is why Nanocubes was chosen to visualize YP's datasets.

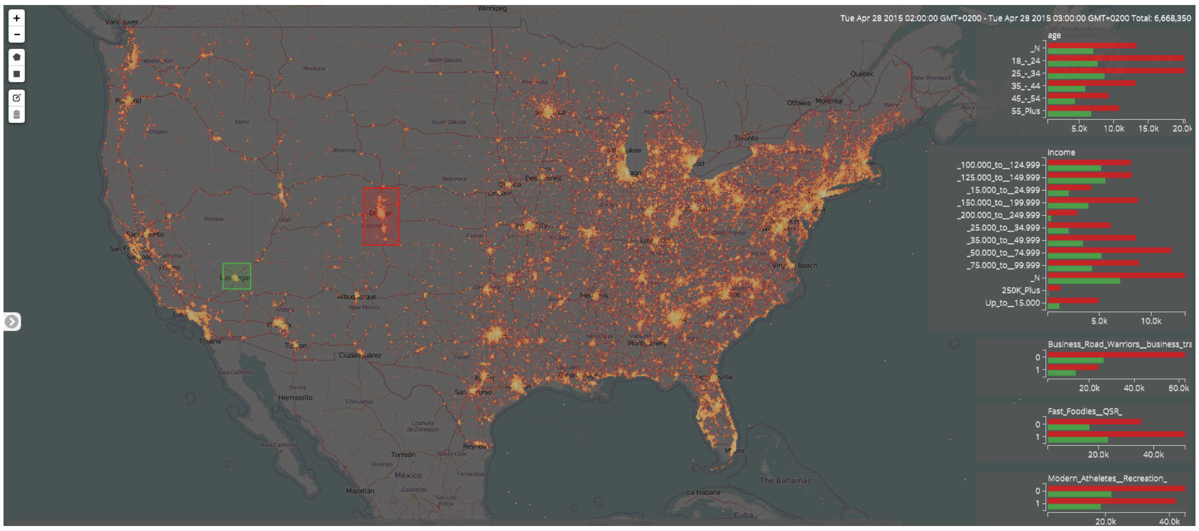


Figure 3.1: Nanocube web client visualization of 6.9 million data points. Comparing Denver with Las Vegas in terms of age, income and many more categories.

3.2 Objective

The currently[2] official nanocube C++ implementation is hosted on GitHub[3]. A new unofficial implementation of nanocubes exists, which eliminates inconveniences that came with the old template heavy C++ implementation. The official code needs to be compiled separately for every nanocube variation of interest e.g. different numbers of categories. Moreover, the new code implements an *Optimal Nanocube Insert Algorithm*, which fixes a redundance imperfection in the old build algorithm by inspecting the quadtree[4] insertion path more precisely for nodes that can be shared or must be copied. Saving and loading of built nanocubes is now also possible with the new code.

Nevertheless, the new C++ implementation had limitations that are tackled and solved in this thesis:

[1]A spatiotemporal data point is composed of a position in space and a point in time
[2]September 9, 2016
[3]https://github.com/laurolins/nanocube
[4]see chapter 4 Nanocubes

- The build time of a nanocube can be tedious.
- Loading built nanocubes is protracted due to ASLR[5].
- Saving & loading built nanocubes does not support compression.
- 32-bit systems are not supported.
- Windows is not supported.

Solving the first two points does facilitate the everyday usage and increases the productivity when working with nanocubes decisively. Solving them is the principal part of this document. The first point is addressed in chapter 5 Multi-part Nanocubes, which describes how to split up a nanocube, calculate the parts concurrently and merge their query results. The vast speedup in build time is shown in section 5.3 Benchmarks. The second point is addressed in chapter 6 Save and Load, in which coincidentally point four gets solved as well (see section 6.2 32-bit Support). Section 6.1 Compression tackles point three. Lastly, chapter 7 Nanocubes on Windows addresses the remaining limitation on this list.

[5] "Address space layout randomization (ASLR) is a computer security technique involved in protection from buffer overflow attacks." - https://en.wikipedia.org/wiki/Address_space_layout_randomization

4 Nanocubes

This chapter is an introduction to the nanocube technology. Nanocubes is based on a client-server model. The server builds up the nanocube data-structure and performs queries from the clients on it. Nanocubes comes with a web client written "in Javascript, HTML5, SVG, WebGL, and D3"[1], which is publicly available. Another nanocube client is being written in C++ by Lauro Lins. It "uses OpenGL for efficient rendering"[2], but is at this point in time[3] not released to the public.

François-Xavier Pineau wrote a YouTube comment[4], in which he couched the idea of nanocubes into one single sentence:

> "Nanocubes is a data structure storing at various spatial resolutions precomputed sets of dense cumulative histograms."

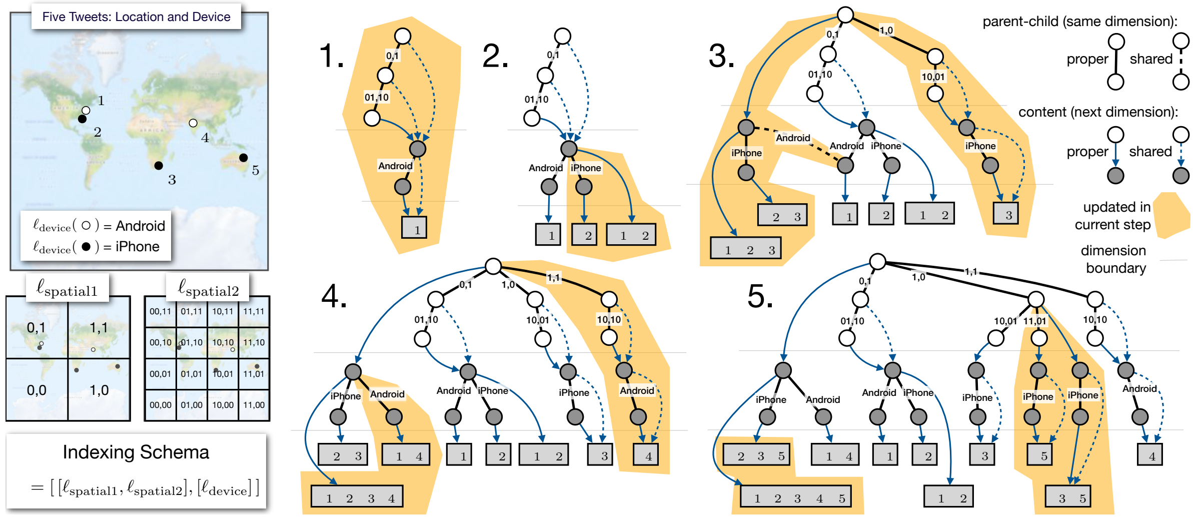

Five Tweets: Location and Device
1
2
3
4
5
$\ell_{\text{device}}(○)$ = Android
$\ell_{\text{device}}(●)$ = iPhone
ℓ_{spatial1}
ℓ_{spatial2}
Indexing Schema
$= [[\ell_{\text{spatial1}}, \ell_{\text{spatial2}}], [\ell_{\text{device}}]]$
1.
2.
3.
4.
5.
parent-child (same dimension):
proper
shared
content (next dimension):
proper
shared
updated in current step
dimension boundary
Android
iPhone

Figure 4.1: "An illustration of how to build a nanocube for five points [o1,...,o5]" [5]

The above figure from the nanocubes paper helps to understand this summarization. It illustrates how a nanocube gets built up by inserting point after point. The five points given have a geological position and a mobile phone operating system, either Android or iOS,

[1][LKS13] Lauro Lins, Jim Klosowski, and Carlos Scheidegger (2013): Nanocubes for Real-Time Exploration of Spatiotemporal Datasets, IEEE InfoVis 2013, p. 6, sect. 5 Implementation
[2]cf. fn.footnote 1
[3]September 9, 2016
[4]https://www.youtube.com/watch?v=8P9QA6TJwys
[5][LKS13] Lins, Klosowski, and Scheidegger (2013), fig. 2, p. 2

associated to them (see the world map in the upper left corner). The nanocubes discussed in this document have all a spatial dimension, which is represented in the form of a quadtree. The two boxes on the left, labeled l_{spatial1} and l_{spatial2}, show the basic idea of a quadtree in this context. A world map gets divided into smaller and smaller squares with each level of resolution. Squares resp. child nodes in the quadtree, which do not contain points, are left out to save storage space. The white nodes in the graphic of the build process represent the quadtree. The top node is called root node and represents a square holding the whole world map. Each node can have up to four child nodes. With each step down the quadtree, the position on the world map is more precisely circumscribed. Usually twenty-five levels are used to reach a sensible resolution. Each node of the quadtree can be asked for its content. In the visualization of the build process, a blue arrow points to the content of a quadtree node. The content of a quadtree node is a root node of another tree structure, which represents the categorical dimension. This second tree structure can be referred to as a *flat tree*, because it always has a depth of one. It consists of a root node with as many child nodes as there are points with different categories associated to the parent quadtree node. Flat tree nodes have content, too. It is a time series of the points that match the position circumscribed by the placed over quadtree node as well as the categorical classification defined by the parent flat tree node. Nodes in both tree structures share content between them to avoid redundant copies of the same content (see dotted arrows). Note that the content of a parent node is always the summarization of the content of its child nodes. This enables very fast querying of the data structure, because there are precomputed time series for every resolution in the spatial as well as in the categorical dimension. The time series are stored as "a sparse variant of summed-area tables"[6], which minimizes the memory footprint and enables fast computation methods to determine the number of points in a specific time span. Read section 4.3.3 Temporal Queries of the nanocube paper[7] for more details.

4.1 Building a Nanocube from raw data

To build a nanocube data-structure, raw data points need to be converted into a particular format first. The file format is called DMP (dump file) and is basically a pre-aggregation of equivalent data points stored in a space efficient binary way. For example, a DMP file generated from Chicago crime statistics can look like this:

```
name: crime50k.csv
encoding: binary
metadata: location__origin degrees_mercator_quadtree25
field: location nc_dim_quadtree_25
field: crime nc_dim_cat_1
valname: crime 7 CRIM_SEXUAL_ASSAULT
valname: crime 13 KIDNAPPING
...
```

[6] [LKS13] Lins, Klosowski, and Scheidegger (2013): p. 6, sect. 4.3.3 Temporal Queries
[7] cf. fn. footnote 6

```
valname: crime 11 INTERFERENCE_WITH_PUBLIC_OFFICER
valname: crime 17 NON-CRIMINAL
metadata: tbin 2013-12-01_00:00:00_3600s
field: time nc_dim_time_2
field: count nc_var_uint_4

AB 3B 83 00 98 B4 41 01 10 8A 00 01 00 00 00 A4 49 83 00 ... (binary data)
```

The first part of the file is separated from the second part by a blank line. It is a textual description of the data set including its name, record encoding type, nanocube field descriptions, value names of the categories with a numeric mapping, and metadata e.g. the point in time to which the encoded time differences of the data points refer to. In this example, the nanocube has four fields: "location nc_dim_quadtree_25" a quadtree with twenty-five levels for the spatial dimension, "crime nc_dim_cat_1" a categorical dimension with up to 256 categories (one byte), "time nc_dim_time_2" a temporal dimension with two bytes to represent the time difference between the timestamp from the metadata and the records, "count nc_var_uint_4" four bytes to count the number of points of the same kind by this measure (position, category, time).

The second part of the file holds the binary encoded records and has the following structure: Firstly, the position in a square grid with $2^n x 2^n$ cells[8] stored in four bytes for each of the two coordinates. As mentioned above, one byte is used to determine the category to which a record belongs, followed by two bytes for the time dimension and four bytes that hold the number of points, that meet the same criteria. All cohesive bytes are stored in the little-endian format. In this example every data packet has a length of fifteen bytes and can be illustrated like this: `| x  | y  |c|t | n  |`.

A Python script called *nanocube-binning-csv* converts comma separated files (CSV) to DMP. Instructions can be found on the official Nanocube GitHub webpage[9]. Note that the CSV file needs to be in chronological order, if the file is read into multiple chunks (default chunk size is 50000 rows but can be adjusted). Otherwise, the resulting DMP file will be rejected while building up the nanocube structure, because the current implementation of the time series does only support in-order insertion.

A DMP file can be read by the nanocube server program. There are two options, either storing the resulting nanocube structure for later use or starting up a query server directly. For example, if *nanocube* is the name of the nanocube server program and *data.dmp* is the dataset of interest, the command lines would look like this:

```
nanocube -d data.dmp -o data.nc
nanocube -l data.nc -q 29512
nanocube -d data.dmp -q 29512
```

The argument -o (output) defines where to store the nanocube, which then can be loaded by using the -l (load) option. The parameter -q defines the query port of the server. New data points can be ingested into a loaded nanocube by streaming in a DMP file via standard input (stdin).

[8]This describes the position on a world map, see quadtree idea in chapter 4 Nanocubes
[9]https://github.com/laurolins/nanocube

```
nanocube -l data.nc -o data.nc < newData.dmp
nanocube -l data.nc -q 29512 < newData.dmp
```

Keep in mind that newData.dmp must only contain time-wise newer data points than the ones already contained in the nanocube (data.nc), otherwise insertion will fail. Moreover, the current implementation does not support *quadtree partitioning*[10] when reading from stdin, because this kind of stream does not support seeking to a defined position, which is necessary to read in a random set of data points without reading in the whole DMP file upfront.

[10]see section 5.1 Quadtree Partition

5 Multi-part Nanocubes

The build time of a nanocube prolongs with every additional dimension. The time needed to build a nanocube is proportional to its size. The size grows exponentially with the number of dimensions, because it is proportional to the number of “product bins” a dataset hits. The product bins can be seen as the Cartesian product of the dimension sets. “For example if a “device” dimension has a “bin" called “iPhone”, and a dimension “language” has a “bin" called “english”, then a record with “iPhone” and “english” values will hit the product bin “(iPhone, english)”. This product bin will need to be represented in the data structure to account for such record.”[1] Adding another categorical dimension does multiply the number of product bins by the cardinality of the additional category set.

YP was interested in building a nanocube with twelve categorical dimensions. Inserting two hundred million data points would have taken approximately two weeks on a modern high end server processor (CPU) with sixteen cores. With respect to the time required to build the nanocube, it was perturbing to see that just one single CPU core was in use. The desire arose to make use of the entire available computing power and thereby diminish the build time to under two days. To satisfy this desire, a nanocube would need to be split up into parts, which then can be built (preferably) independently in parallel. To get the exact same query results that a single parted nanocube would return, the parts themselves or their query results must be merged. Merging two nanocubes resp. parts implies traversing at least one structure entirely, which would be an expensive operation on big datasets. Pseudo code based on the original nanocube pseudo code[2] was developed to examine this approach. See section 9.1 Pseudocode: Merging two Nanocubes in the appendix. Merging the query results turned out to be easier to implement and the introduced overhead in query processing time is negligible as shown in section 5.3 Benchmarks.

5.1 Quadtree Partition

In order to implement a multi-parted version of nanocubes, a sensible way to cut the structure into parts needed to be found. As mentioned in chapter 4 Nanocubes, the nanocube structure relies on quadtrees to efficiently represent the spatial dimension. Without partitioning the quadtree structure, the size of a multi-parted nanocube would grow substantially with every additional part[3], because implementing the sharing of nodes between parts would be to effortful. The opposite effect can be achieved by defining spatial partitions. It is more likely

[1]Lauro Lins: https://github.com/laurolins/nanocube/issues/30#issuecomment-106112543
[2][LKS13] Lins, Klosowski, and Scheidegger (2013): fig. 3, p. 3
[3]compare memory usage in section 5.3 Benchmarks

that nodes can be shared within a spatial focused dataset, which implies that fewer time series need to be stored. Figure 5.1 illustrates this effect.

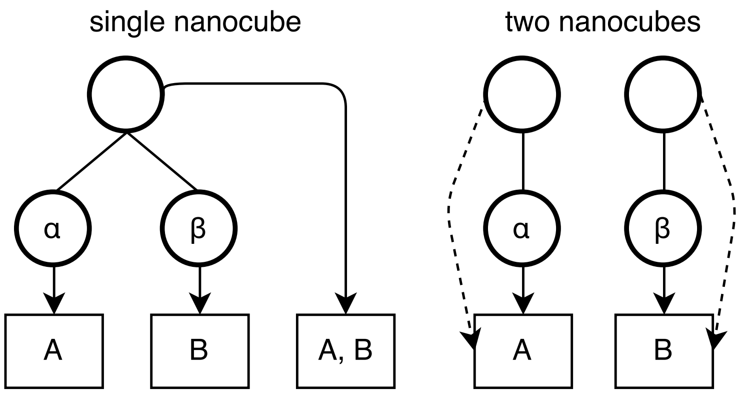


Figure 5.1: Multi-part nanocubes can be more space efficient, when spatial partitions are defined. Sharing is more likely to occur in a spatial focused portion of a dataset.

Since every dataset can have different spatial emphases, the split points of the quadtree need to be computed individually to evenly balance the data points and therefore the computational work over all parts/threads/cores. Precomputing the whole quadtree would be a tedious process. Instead x (default is 10000) random samples of the dataset are inserted into a *count quadtree*, whose nodes count the number of points passing through them during insertion. After insertion, the split points can be determined by adding up the counts of the leaf nodes from left to right until the limit of points for each group/thread/part is reached. For instance, if eight parts are going to be created and 10000 randomly selected samples are inserted, each part should cover about 1250 samples. The more samples taken, the more consummate the split points will be to the actual dataset.

Figuring out the most sensible borders between the groups can be a tricky business. The first case is easy: If adding the count of the leaf node in question does not exceed the threshold of the current group, take it in. If the threshold would be exceeded but the group is empty, take it in regardless. If the group already holds nodes, minimize the overlap by comparing the distances to the threshold with and without the new count added. A smaller distance determines if the count is taken in or added to the subsequent group.

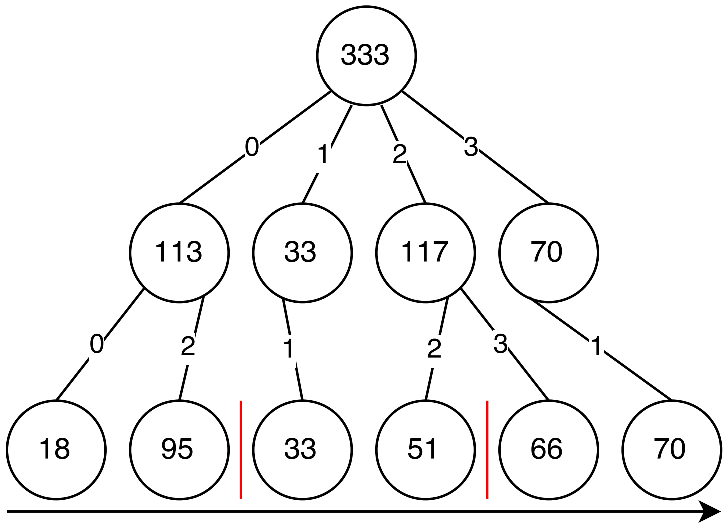


Figure 5.2: Count quadtree: To find three balanced partitions, add the counts from the leaf nodes from left to right while trying to get as close as possible to 111 counts per partition. The two most sensible split points in this example are the nodes reached by taking the quad tree paths 0,2 and 2,2.

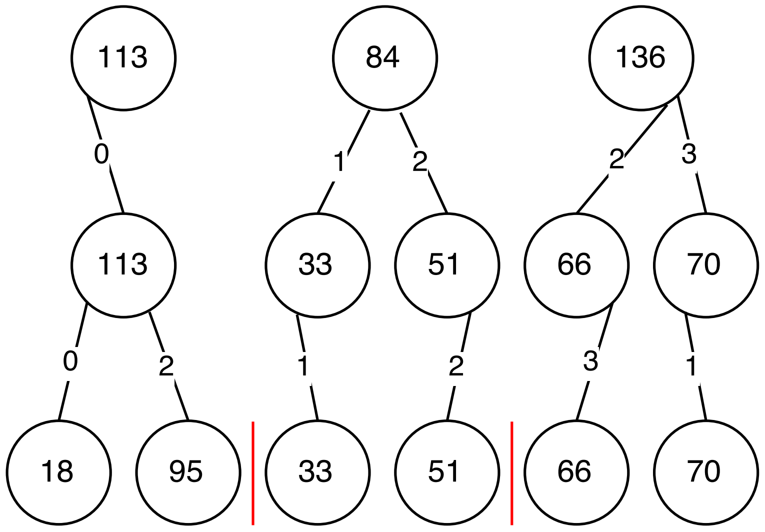


Figure 5.3: Count quadtree (fig. 5.2) partitioned into three parts.

This is the algorithm implemented in C++11. The function gets called with every leaf node (Key) of the *count quadtree* from left to right. The split points are stored in a vector[4] of quadtree addresses. The whole quadtree partitioning project is hosted as a separate project on GitHub[5].

Listing 5.1: A part of the quadtree partitioning C++11 implementation. The function calculates the most sensible borders for a given number of partitions.

```cpp
template <typename Key>
void PartitionFunction<Key>::push(const Key& key, Count count) {

    if (split_points.size() == (std::size_t)num_parts-1)
        return;

    auto threshold = total_count / num_parts;
    if (group.count + count < threshold) {
        group.last_key = key;
        ++group.num_keys;
        group.count += count;
    }
    else {
        if (group.num_keys == 0) {
            split_points.push_back(key);
        }
        else {
            auto diff_without = threshold - group.count;
            auto diff_with    = group.count + count - threshold;
            if (diff_without < diff_with) {
                group.count = count;
                group.last_key = key;
                group.num_keys = 1;
            }
            else {
                split_points.push_back(key);
                group.count    = 0;
                group.last_key = key;
                group.num_keys = 0;
            }
        }
    }
}
```

[4] A vector in C++ is similar to an array in other programming languages. It is a sequence of objects of the same type.

[5] https://github.com/Pyroluk/quadtree_partition

5.2 C++11 Implementation

Besides this document and a few comments in the program code, both C++11 implementations of Nanocubes are not documented. The Unified Modeling Language (UML) class diagram in the appendix[6] helps to understand the program and which modifications where made to implement multi-parted nanocubes.

Like every C++ program, the function called *main* is executed first. The two execution paths inside this function are either loading a nanocube from file(s) or building a new one based on a DMP[7] file. Both paths end up calling a *run* function, which is declared and implemented inside the main function. *run* creates a *Kernel* object, whose main purpose is to store references to the actual nanocube object instance(s), the corresponding nanocube schema[8] and the options entered by the user in the command line. Depending on the options, a built or loaded nanocube is then saved to disc or a server is started to serve queries on the nanocube. In both cases an object called *NanocubeIngest* is used to build a new nanocube or add new data points to a loaded nanocube. The method *run_async* from *NanocubeIngest* starts and returns a C++11 thread instance, which executes the method *run* from *NanocubeIngest*. It reads in a DMP stream point by point either from standard input (stdin) or a file, parses the binary data into an address object (quadtree path) and the variables, which are associated to the point, into a vector object. The address and variables are then passed to the *insert* method of the *Nanocube* object(s).

NanocubeIngest, *Kernel*, *Options* and the *main* function are the first program parts that palpably come to mind as a starting point to upgrade the code to concurrently handle multiple nanocube object instances.

5.2.1 Multi-part command line options

The command line options are extended with two new parameters: `nanocube_parts: -p` and `training_size: -x`. The first option can be used to define how many[9] parts the nanocube should have and if the quadtree partition algorithm should be used. The second option can be used to define how many[10] random data point samples should be read in to build up the *count quadtree* described in section 5.1 Quadtree Partition.

```
nanocube -d data.dmp -o data.nc -p 2
nanocube -d data.dmp -o data.nc -p auto
nanocube -d data.dmp -o data.nc -p auto8
nanocube -d data.dmp -o data.nc -p auto8 -x 100000
nanocube -d data.dmp -o data.nc -p qtpart(20133212103132,21023233231112)
```

The first command line builds a nanocube with two parts. The data points are assigned to the parts alternating one after the other in the order they are stored in the DMP file, without partitioning the quadtree in an elaborated way. The second command line builds a nanocube

[6]section 9.2 UML class diagram
[7]see section 4.1 Building a Nanocube from raw data
[8]see the first part of a DMP file in section 4.1 Building a Nanocube from raw data
[9]default is 1
[10]default is 10000

with as many parts as the CPU has cores and enables quadtree partitioning, too. The third line is similar but concretely sets the number of parts to eight. In addition, the fourth command sets the number of training points to 100000. The last line creates a nanocube with three parts by stipulating two quadtree split points. The stated quadtree addresses are represented as a sequence of child node labels and must have the same length as the nanocube quadtree has levels. The main purpose of this option is the ability to add new data points via stdin to a loaded, quadtree partitioned nanocube. Automatically calculated split points are written to the console via standard out (stdout) and should be noted if new points are going to be added later on. Note that every part is saved into its own separate file. The last command line would result in the creation of three nanocube files: data.0.nc, data.1.nc and data.2.nc. Read chapter 6 Save and Load for an explanation of this design decisions. Starting a server instead of storing the structure to disc works alike. Just exchange `-o data.nc` with e.g. `-q 29512` to define a server port.

5.2.2 Multithreading

The *main* function is adjusted to create a vector of *Nanocube* objects with as many instances as defined by the newly introduced -p parameter. The function arguments and interfaces used in the call hierarchy described above are adapted accordingly. Other minor parts needed to be adjusted too, but they are not in the direct scope of implementing multi-part nanocubes.

NanocubeIngest's range of functions is extended with new tasks. Instead of just inserting point after point into a single nanocube object instance, the *run* method now coordinates the distribution of points to multiple threads, which are then inserting the assigned points into their nanocube part.

If quadtree partitioning is activated, the new method *getPartitionFunction* creates and returns a *PartitionFunction* object with either newly calculated or user-defined split points. The partition function is then used to determine the part of the nanocube to which a given data point belongs. As described in section 5.1 Quadtree Partition, x random data points must be read to calculate sensible borders between the nanocube parts. Since the size of a single record inside a DMP file is known, the exact position of every record inside a DMP file stream can be calculated easily: starting position of binary data + record number * record size. Given the positions in the stream, a large quantity of random samples can be read in expeditiously by using the seek functionality of file streams[11].

Each thread has a queue of data points to process. The queues get filled by the thread that executes the *run* method of *NanocubeIngest*. Reading all points into the queues at once would take up a considerably amount of memory when working with big datasets. Instead, every queue gets filled up every thirty seconds to a dynamically calculated threshold of initially 100000 points. This is done to maximize the insertion speed by minimizing context switches and expensive mutex locks. After calculating the first measurement of how many points per second are processed, the threshold automatically adjusts to the number of point what will approximately be processed until the next refill, plus a buffer of 25%. Because the complexity of inserting a point into a nanocube rises with its size, the number of points

[11]Standard input streams (stdin) do not support seeking, thus quadtree partitioning does not support stdin.

inserted per second shrinks accordingly over time. Therefore, the queues should never empty out before the next refill. Nevertheless, external influences e.g. other programs could cause an unexpected speedup by releasing computational power, which even exceeds the buffer and thus leads to empty queues and a temporary stopped insertion process. This uncommon case is resolved by falling back to the initial threshold if zero points per second are inserted.

The C++11 Standard Library does not provide a ready to use thread-safe queue. Fortunately, Juan from Juan's C++ Blog published a post[12] in which he describes how to add thread-safety to a normal C++11 queue by using a mutex and a condition variable. In a nutshell, "The mutex prevents concurrent reads and writes, and the condition variable allows consuming threads to wait for elements to be available in the queue without excessive mutex contention and without using expensive and inefficient polling."[13]. To keep track of the insertion progress, two variables are added to the provided wrapper class to count the pops and pushes, which the method *getObjectCount* uses to calculate the remaining points in the queue. Moreover, the class is extended with a *hasElement* method. It is a thread-safe wrapper for the *empty* method of the underlying C++11 queue object.

The constructor of *NanocubeIngest* now creates, initiates and stores a thread and a queue for every nanocube part.

Listing 5.2: NanocubeIngest constructor creates threads and queries for parallel data insertion

```
NanocubeIngest::NanocubeIngest(Kernel& kernel,
                               std::istream &input_stream) :
kernel(kernel), input_stream(input_stream) {
    for (auto i = 0; i < kernel.nanocubes.size(); ++i) {
        queues.emplace_back(new ThreadsafeQueue<AddressVariables>());
        threads.push_back(std::thread(&NanocubeIngest::threadInsert,
                          this, i));
    }
}
```

Note that *ThreadsafeQueue* has a C++11 mutex member variable, that cannot be copied or moved, thus making the whole class not copy- nor movable. Because of that, creating a vector of type *ThreadsafeQueue* is tricky. Vector elements must be at least movable by definition. This is achieved by creating a vector of type *unique pointers* (unique_ptr) of type *ThreadsafeQueues*, which are movable:

Listing 5.3: Defining a set of ThreadsafeQueues

```
vector<std::unique_ptr<ThreadsafeQueue<AddressVariables>>> queues;
```

Usually *push_back* is used to insert elements into a vector, but the method creates and inserts a copy of the passed variable and thus cannot be used. *emplace_back* does the trick by constructing the object instance directly into the vector without a copy or move operation. *AddressVariables* is a structure holding the quadtree address and the associated variables of a data point.

[12] https://juanchopanzacpp.wordpress.com/2013/02/26/concurrent-queue-c11/
[13] cf. fn. footnote 12

The threads created and started in the constructor execute the *threadInsert* method of *NanocubeIngest* concurrently. Each thread pops and inserts the data points from its queue until it is empty **and** the thread, that pushes the data into the queues, sets the member variable *done_inserting_queues* to true. Otherwise, the threads will check every one hundred milliseconds for new data in the queue. The pushing thread will wait until every data point is inserted by joining every thread listed in the vector of threads.

Listing 5.4: Every insertion thread executes threadInsert concurrently. They pop data points off their queue and insert them into their nanocube part

```
void NanocubeIngest::threadInsert(int threadNumber) {
    while (!done_inserting_queues ||
           queues[threadNumber]->hasElement()) {
        while (queues[threadNumber]->hasElement()) {
            AddressVariables tmp = queues[threadNumber]->pop();
            kernel.nanocubes[threadNumber]->insert(tmp.address,
                                                   tmp.variables);
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}
```

Besides the threads that insert data points, the vector of threads contains a thread that prints out the current build progress on a terminal by executing the new method *reportStatus*. It prints a progress bar, memory usage in megabytes, time past in seconds, points inserted, points per second, and an estimation of the remaining time in seconds. The estimation does not take into account the deceleration of insertion that occurs with the growth of the data structure. It is simply calculated by dividing the number of remaining points by how many points per second are currently inserted. If the flag `-z` is used in the command line, additional progress bars are shown to indicate the filling level of the queues. The report frequency can be adjusted by the second with the `-f` command line parameter.

```
 58% [============================                        ]
(stdin     ) mem. res:        29MB. time(s):          5
(stdin     ) points inserted: 5898/10000
(stdin     ) points per second: 1179
(stdin     ) est. remaining time(s): 3
```

Figure 5.4: Nanocube build progress report

A major difference between the new and the old, but still official, code is custom memory allocation tailored for nanocubes. The two pure static classes *PoolAllocatorWrapper* and *SimpleAllocatorWrapper* held an instance of *PoolAllocator* resp. *SimpleAllocator*. Every part of the program, that allocates and frees memory within the nanocube structure, called the method *malloc* and *free* of one of the two wrapper classes. The wrappers then just passed the call to the encapsulated allocator instance. The two wrappers got replaced with *SlabAllocatorWrapper*, due to the new allocator implementation described in chapter 6 Save and Load. Unfortunately, all allocator implementations including the new one are not thread-safe. Serializing the memory allocation with normal C++11 mutexes does retard the build process immensely because allocation and deallocation of memory is a very frequent operation during the build process and locking a mutex is time-consuming. The performance impact can be significantly reduced by using a spinlock [14], but the difference without using locks at all is still substantial. For that reason, every nanocube part has its own allocator instance. The static allocation wrapper(s) are deeply intertwined in the code. Amongst other things, they are passed as template arguments in several classes of the program. Changing this design decision would be a project on its own, especially without code documentation. Therefore, the wrapper(s) now holds a vector of allocators, one for each nanocube part. In order to use the correct allocator instance, every class instance using the wrapper(s) needs to know to which part of the nanocube it belongs. The part number is assigned first in the *create* method of the *Nanocube* class, which passes the number to the constructor of the class. The number gets propagated through the whole structure by the methods *allocateRootNode, allocateInternalNode* and *allocateSummaryNode*, which pass the number to the corresponding constructor of the node class in question. The downside of this design is the additional memory space needed to store the numbers[15].

5.2.3 Merging query results

The merging of the queue results of each nanocube part is implemented in the *serveQuery* method of the *NanocubeServer* class. The method gets called from a handler of the nanocube server with the name "count", which gets initialized in the method *initializeQueryServer* of the same class. A *Request* and a *Program* object are passed into the method. The Request object holds a pointer to a *mg_connection* instance from the popular Mongoose Embedded Web Server Library[16] and is at the end of the procedure used to send back the response in text, json or binary format. A request string e.g. count.r("timestamp",interval(648,649)).a("location", dive(tile2d(154,320,9),7),"img") gets parsed by a *Parser* instance inside the handler into a *Program* object, which is a singly linked list of *Call* objects. Inside the *serveQuery* method, the request in form of a *Program* object gets parsed once again. This time into a *Query* object with the method *parse_program_into_query* from the *NanocubeServer* class. Each *Nanocube* object instance creates its own *Query* instance by calling the *query* method of the *Nanocube* class. The query get executed by calling the *execute* method of the *Query* class. The resulting *TreeStore* object is generated by a *TSeriesCollector* object instance, which gets passed in the *execute* call. Since the *Query* objects are depending on the *Nanocube* object instance from which they got created from, every nanocube part's *Query* object must be filled separately with the

[14] "In software engineering, a spinlock is a lock which causes a thread trying to acquire it to simply wait in a loop ("spin") while repeatedly checking if the lock is available. Since the thread remains active but is not performing a useful task, the use of such a lock is a kind of busy waiting. Once acquired, spinlocks will usually be held until they are explicitly released, although in some implementations they may be automatically released if the thread being waited on (that which holds the lock) blocks, or "goes to sleep". Because they avoid overhead from operating system process rescheduling or context switching, spinlocks are efficient if threads are likely to be blocked for only short periods." - https://en.wikipedia.org/wiki/Spinlock

[15] A few megabytes depending on the number of nodes.

[16] https://github.com/cesanta/mongoose

"Program". Generating and executing the queries for every nanocube part is implemented by encapsulating the described calls into *Future* objects, which get created by the C++11 template function *async*. The launch behavior of the futures can be adjusted with a flag in the async call. For now the execution of the futures and therefore of the queries is done sequentially, because a dependency on another project called Polycover is not thread-safe yet. If this problem is fixed, parallel execution of the queries can be enabled by simply replacing *std::launch::deferred* with *std::launch::async* in the async call. The Futures are stored in a vector of Futures and their results are pushed into a vector of *TreeStores*. The *TreeStores* are then merged together into a single *TreeStore* to generate the final query result. For every query type, the tree stored inside a *TreeStore* has always a depth of one. The child nodes of the root are implemented as an unordered_map, which makes merging the trees a straight forward process. The first not empty tree in the vector is the starting point of the merge process. All other trees are merged into it by either adding up the values of equivalent child nodes or by copying over missing child nodes into the merged tree. If the trees only consist of the root nodes, their values are summed up in the root of the merged tree. The unordered_map should not be manipulated directly, because of a reasoned and coherent memory allocation scheme behind the TreeStore structure. For instance, the method *getOrCreateChild* from the *InternalNode* class must be used to create new child nodes. Moving[17] entries between the maps, results in a very hard to find memory error, because the destructor of the *InternalNode* class will try to delete the already moved objects.

Listing 5.5: Executing queries and merging their results

```
std::vector<std::future<tree_store::TreeStore<config_type>>>
                                                    treeStoreFutures;
for (int i = 0; i < num_nanocubes; ++i) {
    treeStoreFutures.push_back(std::async(std::launch::deferred,
                                          [&](int i) {
        auto query = kernel.nanocubes[i]->query();
        parse_program_into_query(program, query);

        if (queryMode == query_type::UNDEFINED)
            queryMode = query.mode;

        TSeriesCollector collector;
        query.execute(collector);

        const tree_store::TreeStore<config_type> &result =
                                            *collector.tree.get();

        return result;
    }, i));
}

...
```

[17]C++11 supports move semantics, which can be used to avoid unnecessary copy operations of objects.

```cpp

std::vector<tree_store::TreeStore<config_type>> treeStores;
for (auto& treeStore : treeStoreFutures)
    treeStores.push_back(treeStore.get());

tree_store::TreeStore<config_type> mergedTree;
int i = 0;
do {
    mergedTree = treeStores[i];
} while (mergedTree.root.get() == nullptr &&
         ++i < treeStores.size());

if (mergedTree.root.get() != nullptr) {
    if (mergedTree.root->isInternalNode()) {
        auto& mergedChilds =
                        mergedTree.root->asInternalNode()->children;

        bool isFirstTreeStore = true;
        for (auto& treeStore : treeStores) {
            if (treeStore.root.get() == nullptr)
                continue;
            if (isFirstTreeStore) {
                isFirstTreeStore = false;
                continue;
            }

            auto& childs =
                        treeStore.root->asInternalNode()->children;
            for (auto& child : childs) {
                bool isNew = false;
                auto foundOrNewChild = mergedTree.root->
                        asInternalNode()->
                        getOrCreateChild(child.first, true, isNew);

                if (isNew)
                    foundOrNewChild->asLeafNode()->value =
                            child.second.node->asLeafNode()->value;
                else
                    foundOrNewChild->asLeafNode()->value +=
                            child.second.node->asLeafNode()->value;
            }
        }
    }
    else //root is leafnode
    {
        bool isFirstTreeStore = true;
        for (auto& treeStore : treeStores) {
            if (treeStore.root.get() == nullptr)
                continue;
            if (isFirstTreeStore) {
                isFirstTreeStore = false;
                continue;
            }

            mergedTree.root->asLeafNode()->value +=
                                treeStore.root->asLeafNode()->value;
        }
    }
}
```

5.3 Benchmarks

In order to test the performance and integrity of the implementation of Multi-part Nanocubes, **nanocubesBenchmark**[18] was written. It is a C++11 program, which measures the time needed to build and query nanocubes. It optionally validates the query results, too. The number of parts is increased with each iteration starting from one up to the number defined with the `-p` parameter. The `auto` and `qtpart`[19] keywords are supported, too. If the nanocube executable is in another folder then the benchmark executable, the file path must be specified with the `-n` parameter. The `-e` flag can be used to set the process priority to high on windows or respectively the nice value to -15 on linux and mac systems. This reduces the influence of other processes on the benchmark results. All other nanocube parameters are supported too and are forwarded to the nanocube program.

The code should compile and run on every platform that supports a C++11 compiler and the Boost C++ libraries[20]. Boost.Asio is used to query the nanocubes over HTTP in JSON mode.

The C++11 Standard Library does not support the creation of processes. Fortunately, Ole Christian Eidheim published “A small platform independent library making it simple to create and stop new processes in C++, as well as writing to stdin and reading from stdout and stderr of a new process”[21]. The end of the insertion process is detected by comparing the strings from the standard output of the nanocube program with the unique string “(stdin:done)”. To prevent a truncation of this important string, the nanocube program is modified to flush the standard out buffer accordingly.

The validation of query results can be enabled by specifying a file path with the `-u` command line parameter to a file containing nanocube queries. This file can either be a plain textfile with one query per line or a json file generated by the NetExport[22] extension for the popular

[18]https://github.com/Pyroluk/nanocubesBenchmark
[19]see section 5.2 C++11 Implementation
[20]http://www.boost.org/
[21]https://github.com/eidheim/tiny-process-library
[22]http://www.softwareishard.com/blog/netexport/

Mozilla Firefox Add-on Firebug[23]. With the help of the extension, Firebug can be used to capture every nanocube query sent by the web client. To gather large quantities of queries, the net.logLimit setting of Firebug should be set so zero (disable limit) or an appropriate number of entries (default is 500). The HTTP tracing file contains both queries and results, but the validation is currently based on the results of the first iteration of the benchmark process with only one nanocube part. This behaviour can be changed by uncommenting the code in line 324 of the nanocubesBenchmark.cc source file. This enables validation based on the results stored in the tracing file. The order of child nodes in a responded "tree structure" can be different with every request, because there is no defined execution order of nanocube parts in the merge process. Therefore, to validate a response, the children are read into unorderd_maps to check for mutations like different values and missing or unexpected nodes.

5.3.1 Procedure

The program first reads in the file containing the queries, then parses the command line parameters and determines the filename of the new logfile (testlog0.txt, testlog1.txt, ..., testlogX.txt). Depending on how many parts are going to be tested maximally, in a loop from 1 to x parts, a command line with the current nanocube parameters gets generated, printed out and passed to a newly started nanocube process. If requested, the process priority is adjusted to privilege the nanocube program. A stopwatch is started directly before starting the nanocube process. The standard output from the nanocube process is read and checked for `(stdin:done)` to determine the end of the insertion process, after which the stopwatch is stopped too. The last status the nanocube process printed is written to the terminal and into the logfile, together with the measured time needed for the insertion procedure.

A second stopwatch is started to measure the query test procedure. For each query, a HTTP request string is generated and sent to the nanocube process via a TCP connection. The complete response is read after parsing the content length, if the correct HTTP status code (200, OK) is returned. If enabled, the responded query result is checked for validity against the result responded in the first benchmark iteration with only one nanocube part as described above. The JSON string gets parsed with the help of the library JsonCpp[24]. The number of successful queries is counted and printed out together with the time needed to complete all queries.

Lastly, the nanocube process gets killed programmatically. After waiting two seconds, the next benchmark iteration starts or the benchmark process ends, if no further iterations are left.

5.3.2 Results

The performance was tested on several CPUs ranging from a high end Intel Core i7 CPU with Hyper-Threading, over an AMD Accelerated Processing Unit (APU) down to systems on a chip (SoC) found on single-board computers like the Raspberry Pi and Banana Pi resp.

[23]http://getfirebug.com/

[24]https://github.com/open-source-parsers/jsoncpp

all sorts of mobile devices like smartphones and tablets. All processing units are running a 64-bit version of Windows, except the SoCs, which run 32-bit versions of Raspbian[25].

Test subjects ordered by computing power, fastest first:

- Intel Core i7-4710HQ, 4x 2.5 GHz, 3.5 GHz Turbo, 8 Threads, 64-bit, Windows 8.1
- AMD Phenom II X4 955 Black Edition, 4x 3.4 GHz, 4 Threads, 64-bit, Windows 10
- AMD Phenom X4 9550, 4x 2.2 GHz, 4 Threads, 64-bit, Windows 7
- AMD Athlon 5350 APU, 4x 2.05 GHz, 4 Threads, 64-bit, Windows 7
- Intel Pentium Dualcore E2140, 2x 3 GHz, 2 Threads, 64-bit, Windows 7
- Allwinner A20 SoC ARMv7-A, Banana Pro, 2x 1 GHz, 2 Threads, 32-bit, Raspbian Jessie
- Broadcom BCM2835 SoC ARMv6, Raspberry Pi B rev. 2, 1x 950 MHz, 1 Thread, 32-bit, Raspbian Jessie

Each processor ran four benchmarks in total[26], comparing the performance with and without quadtree partitioning[27] as well as with TCMalloc[28] and without. The maximum number of parts was set to modestly oversubscribe the CPU with one or two more threads than physically present. When using quadtree partitioning, oversubscription often boosted the insertion speed even further, because spatial focused datasets are faster to process into the nanocube data structure.

The measurement series are based on a real world dataset with *eleven categorical dimensions* generated by YP from data gathered from RTB systems[29]. The quadtree(s) have the common depth of twenty-five levels. Only the first 100000 data points of the 6.9 million spanning dataset were inserted to keep the build times in reasonable boundaries while benchmarking. To determine the query speed, the time needed to process 3057 queries was measured. The queries originate from browsing through the dataset with the web client, while using the whole range of functions in a realistic fashion. Note that the measured query times include JSON conversions of the query results.

Splitting up a problem into parts in such a way that they can be solved independently (on separate processor cores) naturally results in a linear speedup, often minus a small overhead. This effect can be observed when quadtree partitioning is not used. The speedup is limited to the number of physical CPU cores. Oversubscription only results in more memory usage, slower query times and often even slower insertion speeds. In contrast, when using quadtree partition, oversubscription lowers the memory usage and speeds up the insertion even further

[25]Raspbian is based on Debian GNU/Linux

[26]The Raspberry Pi was only subject in the second benchmark run with TCMalloc due to the incredibly slow performance

[27]see section 5.1 Quadtree Partition

[28]read section 5.3.3 Profiling: Even more Speed with TCMalloc for details

[29]read section 3.1 Motivation

with every additional thread/part. Surprisingly, not even the query speed is influenced noticeably. Only when using over thirty threads to calculate the nanocube, the query speed slows down by 10 %[30]. Partitioning the quadtree results in a more than linear speedup on every tested CPU. To a certain degree, it even makes sense to highly oversubscribe the CPU cores. As fig. 5.6 Intel Core i7-4710HQ: up to 100 Threads, Insertion Speed shows, the insertion speed keeps rising, but the memory consumption starts to climb again, if (too) many threads are used[31].The query speed is another factor to consider, because it nearly linearly decreases with every additional thread[32]. Table 5.27 TCMalloc Intel Core i7-4710HQ with Quadtree Partition: up to 100 Threads holds the whole measurement series of the oversubscription test.

The "linear" and "TCMalloc linear" lines in the following graphs illustrate the theoretical linear growth in insertion speed. They are both based on the speed measured in the first benchmark iteration with only one thread/part. The insertion speed is measured in points per second similar to the query speed, which is measured in queries per second.

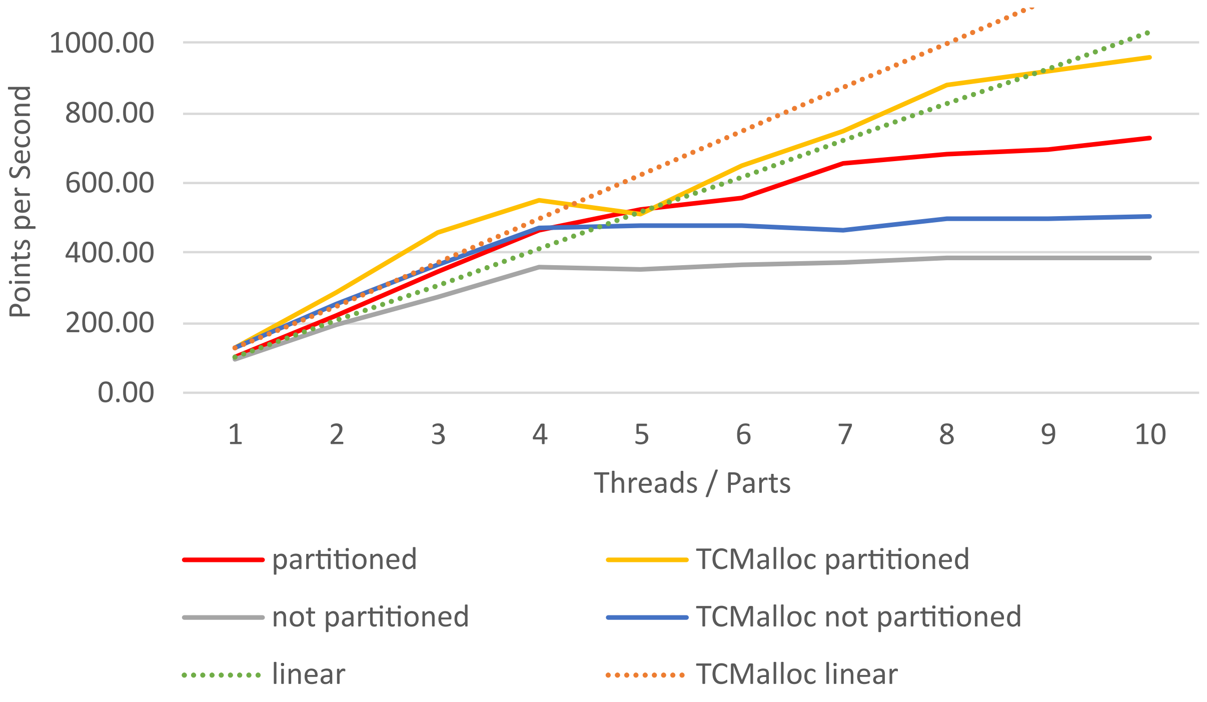


Figure 5.5: Intel Core i7-4710HQ

[30]see fig. 5.8 Intel Core i7-4710HQ: up to 100 Threads, Query Speed and table 5.27 TCMalloc Intel Core i7-4710HQ with Quadtree Partition: up to 100 Threads
[31]see fig. 5.7 Intel Core i7-4710HQ: up to 100 Threads, Memory Consumption
[32]see fig. 5.8 Intel Core i7-4710HQ: up to 100 Threads, Query Speed

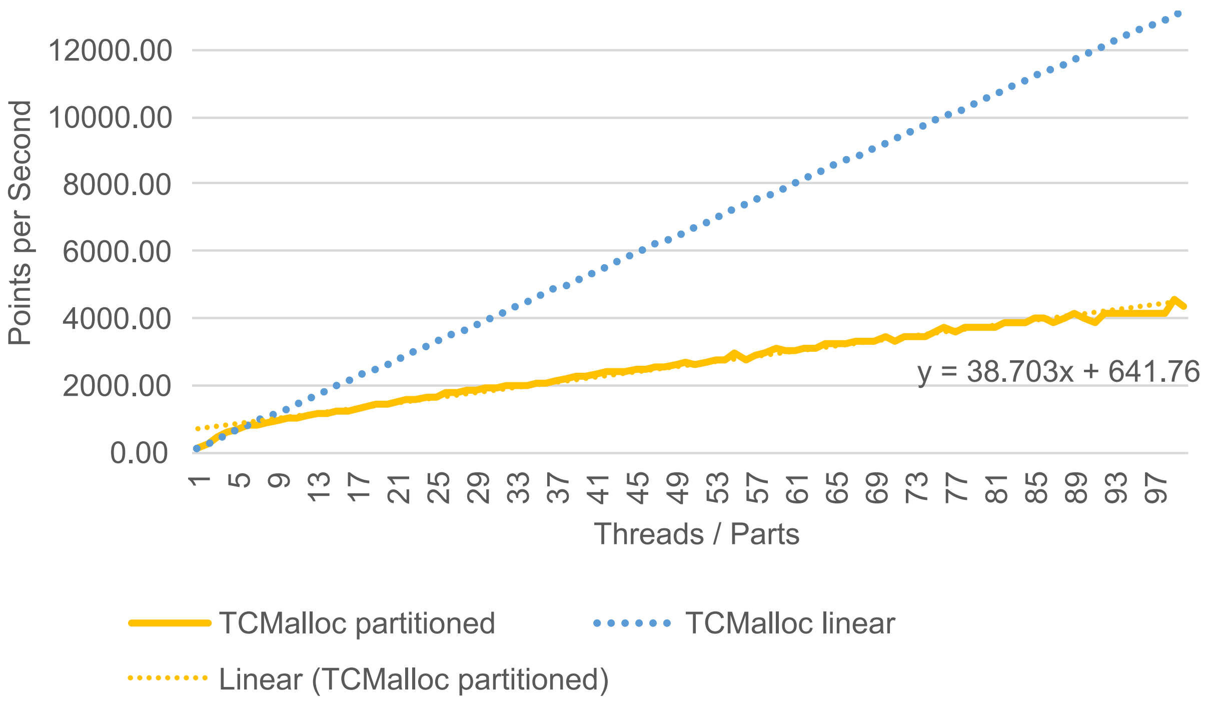


Figure 5.6: Intel Core i7-4710HQ: up to 100 Threads, Insertion Speed

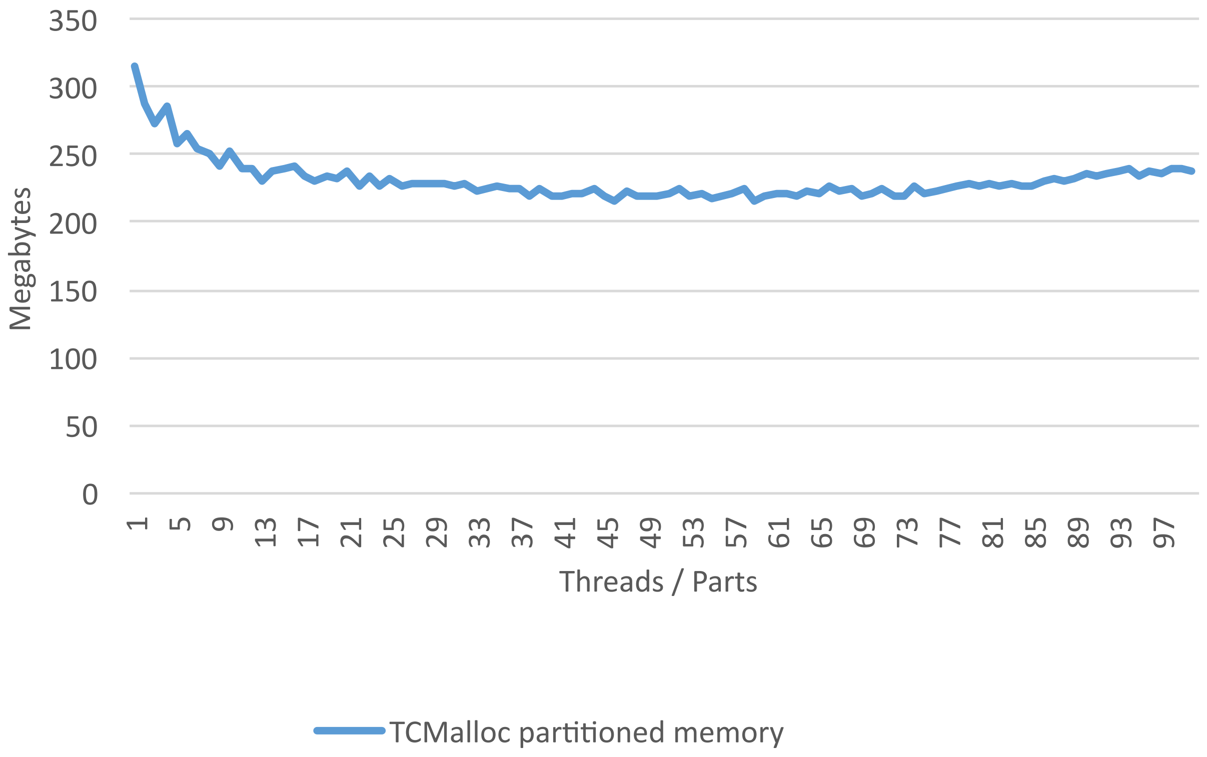


Figure 5.7: Intel Core i7-4710HQ: up to 100 Threads, Memory Consumption

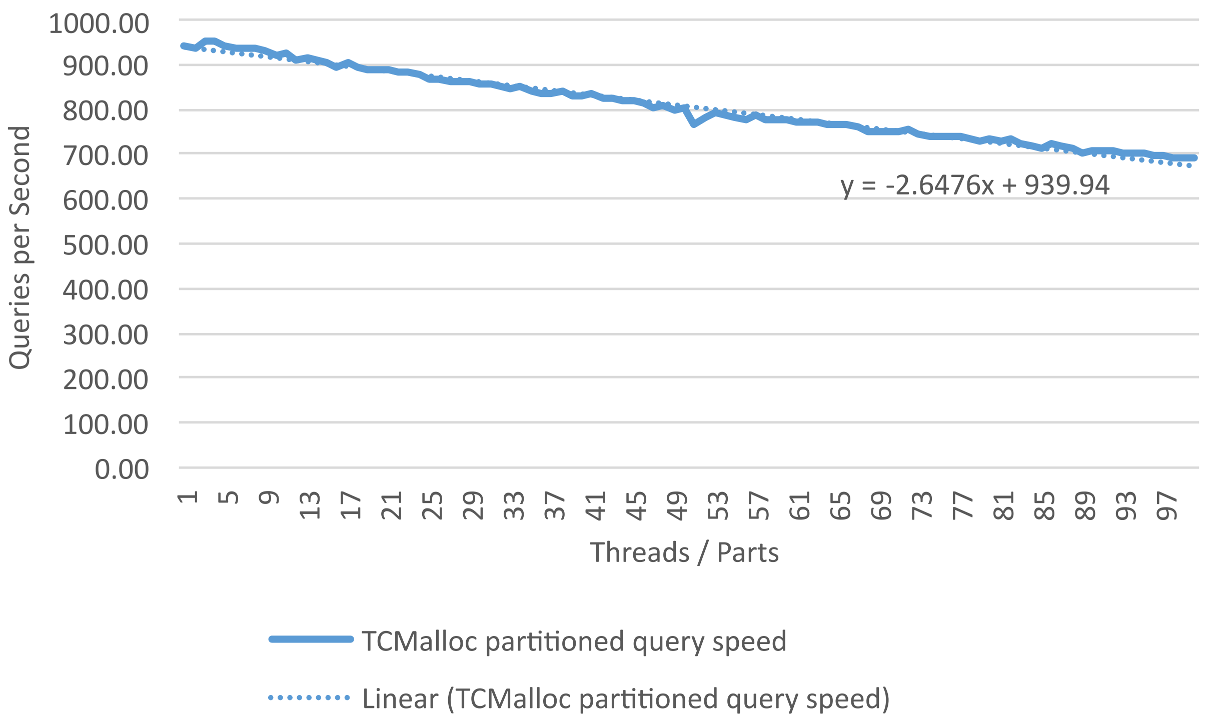


Figure 5.8: Intel Core i7-4710HQ: up to 100 Threads, Query Speed

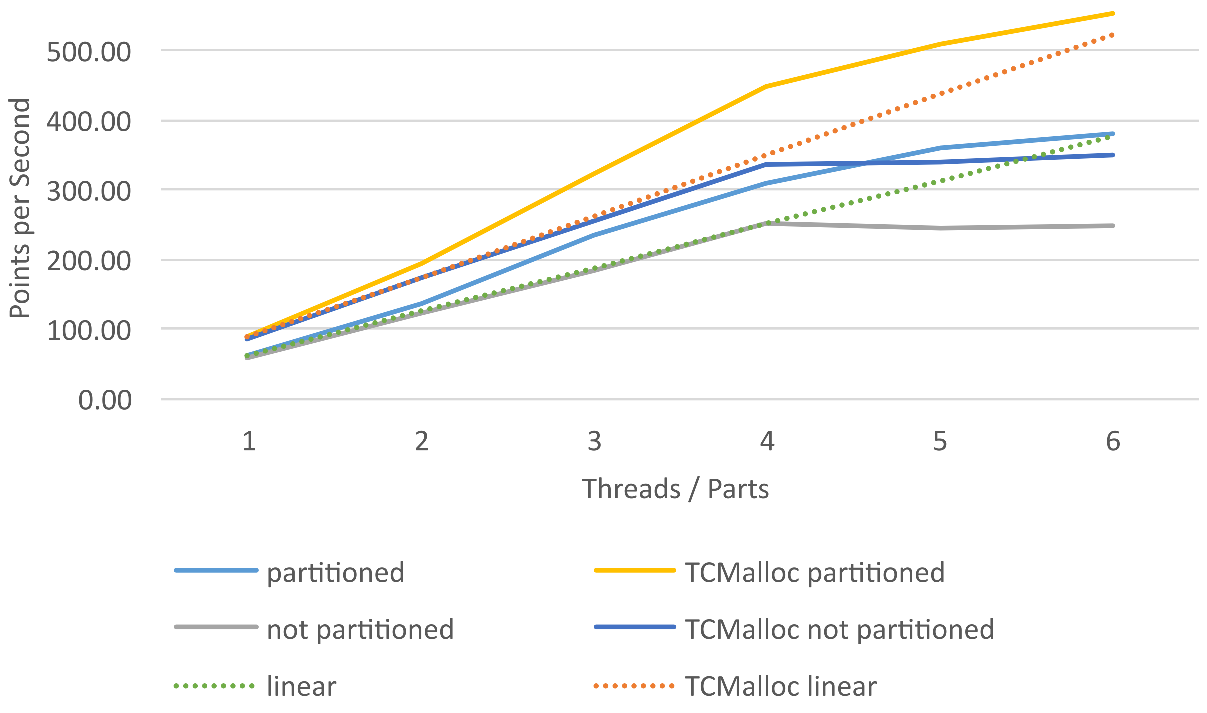


Figure 5.9: AMD Phenom II X4 955 Black Edition

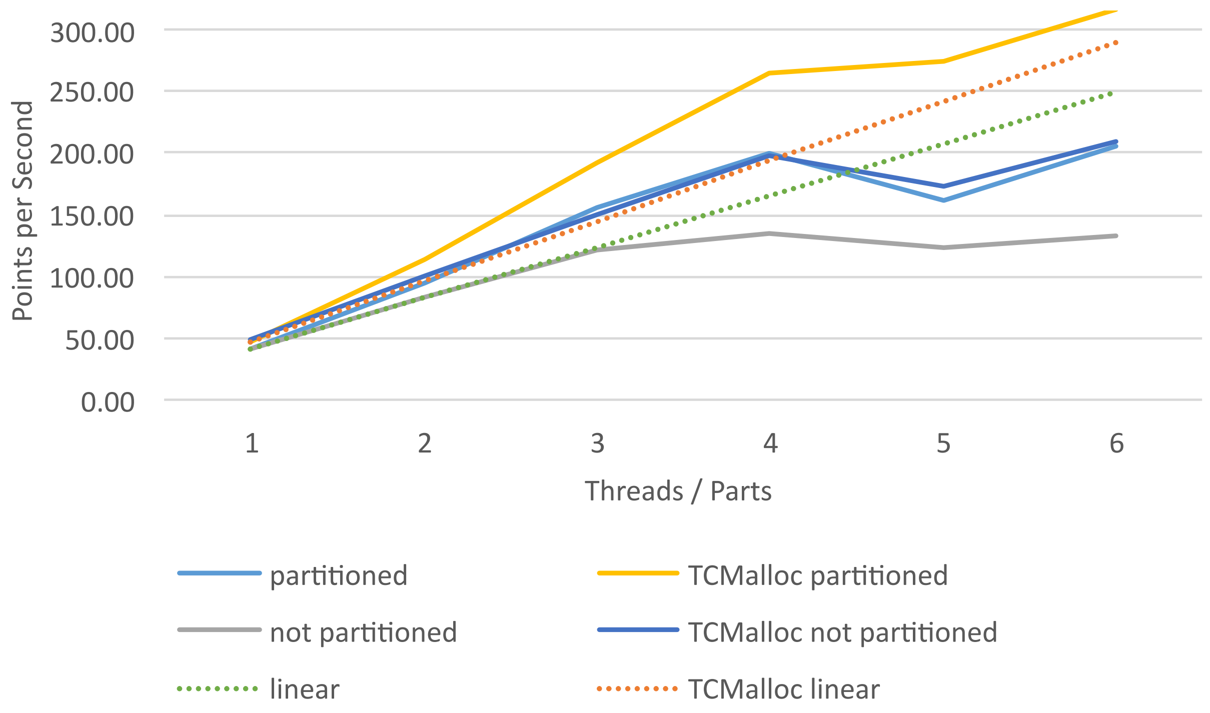


Figure 5.10: AMD Phenom X4 9550

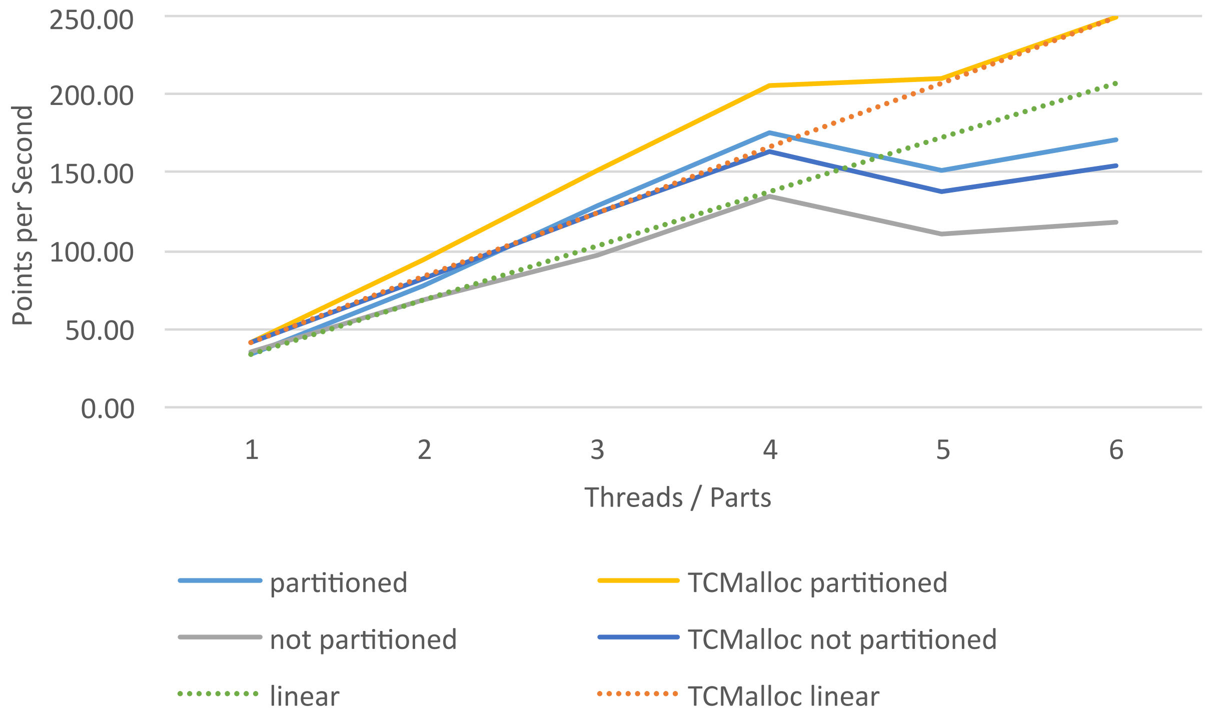


Figure 5.11: AMD Athlon 5350 APU

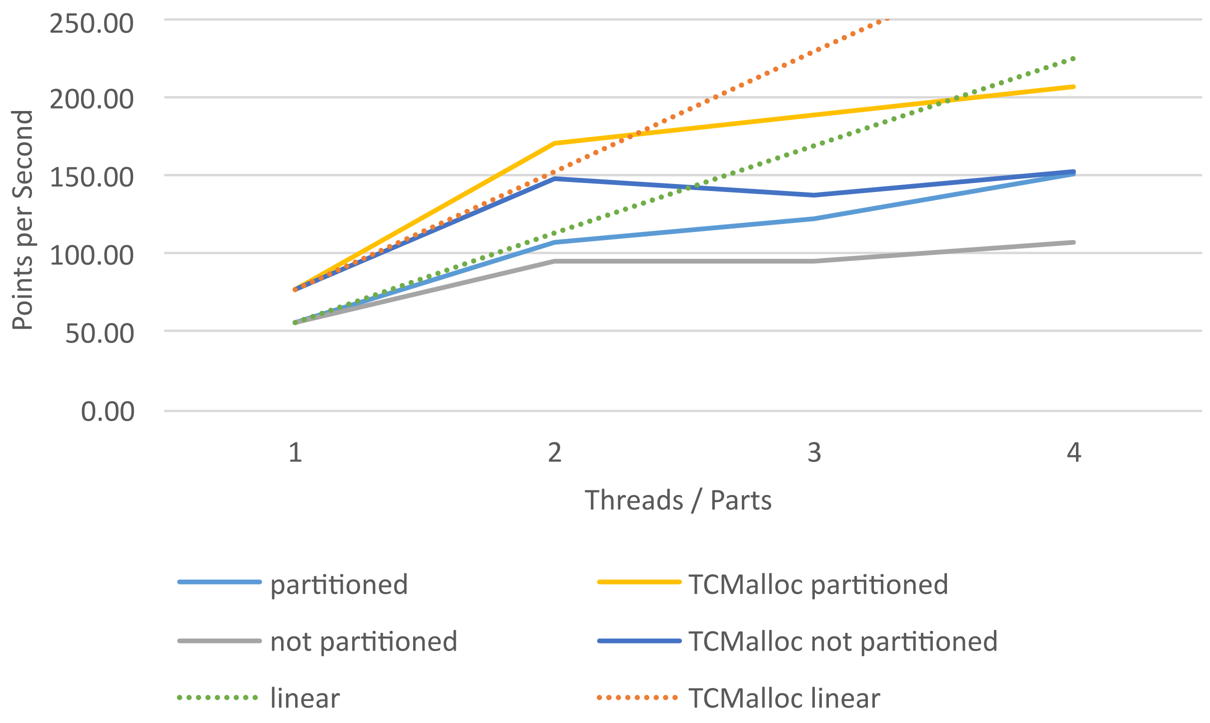


Figure 5.12: Intel Pentium Dualcore E2140

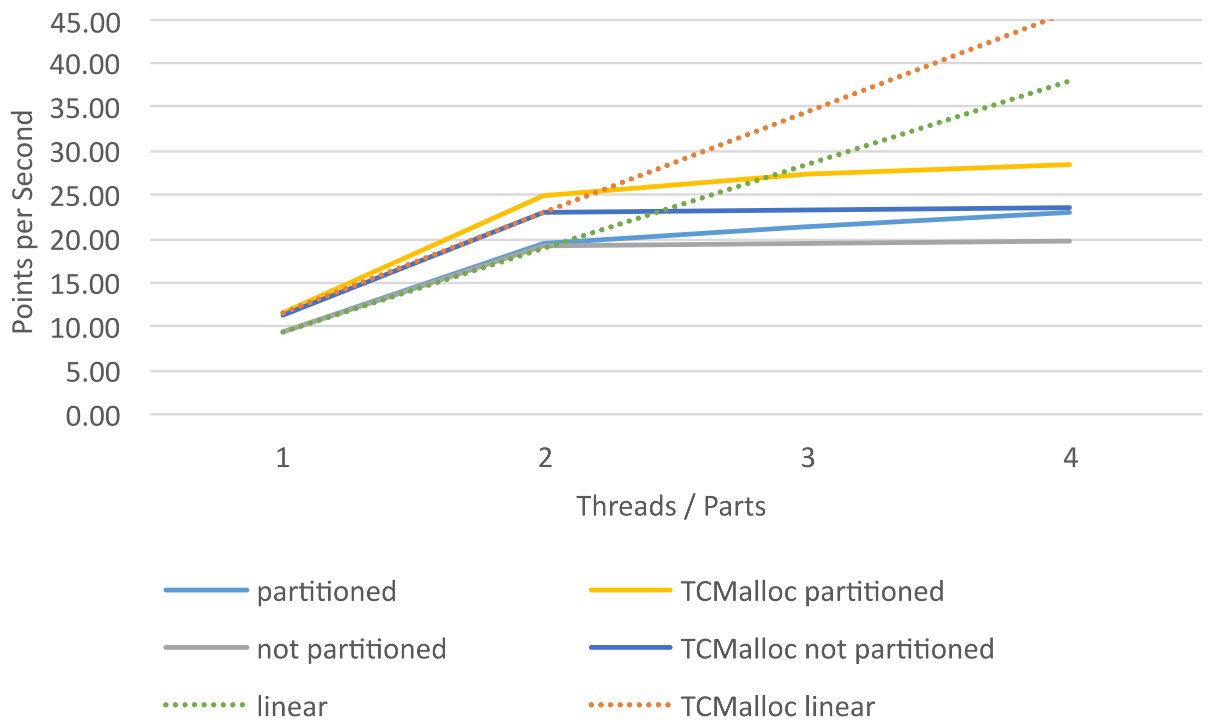


Figure 5.13: Allwinner A20 SoC ARMv7-A, Banana Pro

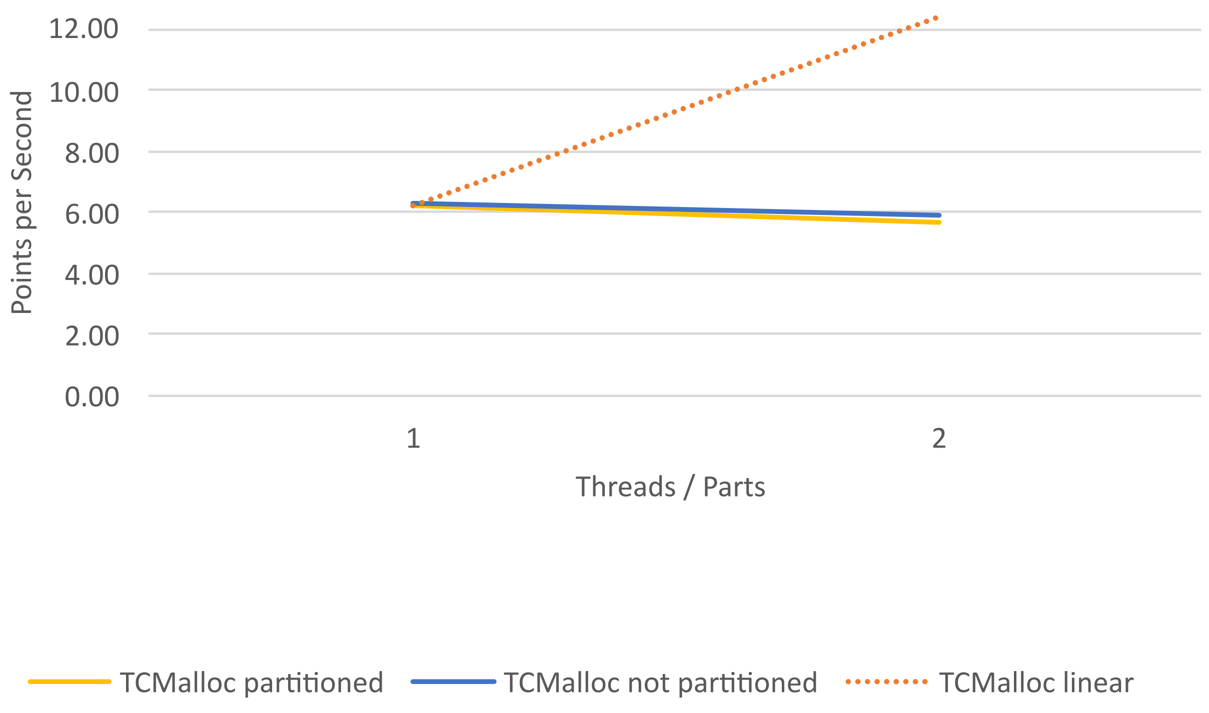


Figure 5.14: Broadcom BCM2835 SoC ARMv6, Raspberry Pi B rev. 2

The “relative difference columns” in the tables relate to the column in front of it. All percentages relate to the measurement from a single parted nanocube (first row). For example, in table 5.1 Intel Core i7-4710HQ with Quadtree Partition the insertion speed is about 700 % faster when using ten threads/parts instead of just one. T. stands for number of threads/parts.

Table 5.1: Intel Core i7-4710HQ with Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	102.99 p/s	971 s	-	312 MB	-	3711 s	823.77 q/s	-
2	219.30 p/s	456 s	212.94 %	282 MB	90 %	3582 s	853.43 q/s	104 %
3	347.22 p/s	288 s	337.15 %	265 MB	85 %	3635 s	840.99 q/s	102 %
4	460.83 p/s	217 s	447.47 %	270 MB	87 %	3665 s	834.11 q/s	101 %
5	526.32 p/s	190 s	511.05 %	249 MB	80 %	3651 s	837.30 q/s	102 %
6	555.56 p/s	180 s	539.44 %	251 MB	80 %	3696 s	827.11 q/s	100 %
7	653.59 p/s	153 s	634.64 %	241 MB	77 %	3717 s	822.44 q/s	100 %
8	684.93 p/s	146 s	665.07 %	246 MB	79 %	3741 s	817.16 q/s	99 %
9	694.44 p/s	144 s	674.31 %	233 MB	75 %	3699 s	826.44 q/s	100 %
10	729.93 p/s	137 s	708.76 %	235 MB	75 %	3748 s	815.64 q/s	99 %

Table 5.2: Intel Core i7-4710HQ without Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	93.98 p/s	1064 s	-	308 MB	-	3623 s	843.78 q/s	-
2	194.17 p/s	515 s	206.60 %	602 MB	195 %	3852 s	793.61 q/s	94 %
3	274.73 p/s	364 s	292.31 %	732 MB	238 %	4082 s	748.90 q/s	89 %
4	361.01 p/s	277 s	384.12 %	868 MB	282 %	4339 s	704.54 q/s	83 %
5	353.36 p/s	283 s	375.97 %	992 MB	322 %	4541 s	673.20 q/s	80 %
6	367.65 p/s	272 s	391.18 %	1187 MB	385 %	4768 s	641.15 q/s	76 %
7	370.37 p/s	270 s	394.07 %	1382 MB	449 %	4984 s	613.36 q/s	73 %
8	383.14 p/s	261 s	407.66 %	1466 MB	476 %	5130 s	595.91 q/s	71 %
9	387.60 p/s	258 s	412.40 %	1449 MB	470 %	5351 s	571.30 q/s	68 %
10	386.10 p/s	259 s	410.81 %	1585 MB	515 %	5439 s	562.05 q/s	67 %

Table 5.3: AMD Phenom II X4 955 Black Edition with Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	62.70 p/s	1595 s	-	316 MB	-	5327 s	573.87 q/s	-
2	134.59 p/s	743 s	214.67 %	284 MB	90 %	5271 s	579.97 q/s	101 %
3	233.10 p/s	429 s	371.79 %	270 MB	85 %	5094 s	600.12 q/s	105 %
4	310.56 p/s	322 s	495.34 %	273 MB	86 %	5161 s	592.33 q/s	103 %
5	358.42 p/s	279 s	571.68 %	260 MB	82 %	5455 s	560.40 q/s	98 %
6	380.23 p/s	263 s	606.46 %	249 MB	79 %	5244 s	582.95 q/s	102 %

Table 5.4: AMD Phenom II X4 955 Black Edition without Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	59.77 p/s	1673 s	-	312 MB	-	5182 s	589.93 q/s	-
2	121.95 p/s	820 s	204.02 %	605 MB	194 %	5542 s	551.61 q/s	94 %
3	185.19 p/s	540 s	309.81 %	736 MB	236 %	5975 s	511.63 q/s	87 %
4	250.63 p/s	399 s	419.30 %	871 MB	279 %	6424 s	475.87 q/s	81 %
5	245.70 p/s	407 s	411.06 %	994 MB	319 %	6657 s	459.22 q/s	78 %
6	246.91 p/s	405 s	413.09 %	1185 MB	380 %	7182 s	425.65 q/s	72 %

Table 5.5: AMD Phenom X4 9550 with Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	41.37 p/s	2417 s	-	319 MB	-	8203 s	372.67 q/s	-
2	95.42 p/s	1048 s	230.63 %	281 MB	88 %	8000 s	382.13 q/s	103 %
3	155.04 p/s	645 s	374.73 %	270 MB	85 %	7843 s	389.77 q/s	105 %
4	200.00 p/s	500 s	483.40 %	269 MB	84 %	8062 s	379.19 q/s	102 %
5	160.77 p/s	622 s	388.59 %	251 MB	79 %	8171 s	374.13 q/s	100 %
6	205.34 p/s	487 s	496.30 %	263 MB	82 %	8312 s	367.78 q/s	99 %

Table 5.6: AMD Phenom X4 9550 without Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	41.44 p/s	2413 s	-	313 MB	-	8187 s	373.40 q/s	-
2	82.64 p/s	1210 s	199.42 %	606 MB	194 %	9000 s	339.67 q/s	91 %
3	122.25 p/s	818 s	294.99 %	738 MB	236 %	9765 s	313.06 q/s	84 %
4	134.95 p/s	741 s	325.64 %	869 MB	278 %	10484 s	291.59 q/s	78 %
5	123.61 p/s	809 s	298.27 %	995 MB	318 %	11015 s	277.53 q/s	74 %
6	132.45 p/s	755 s	319.60 %	1189 MB	380 %	11609 s	263.33 q/s	71 %

Table 5.7: AMD Athlon 5350 with Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	34.40 p/s	2907 s	-	314 MB	-	9578 s	319.17 q/s	-
2	77.16 p/s	1296 s	224.31 %	287 MB	91 %	9401 s	325.18 q/s	102 %
3	128.37 p/s	779 s	373.17 %	271 MB	86 %	9391 s	325.52 q/s	102 %
4	175.13 p/s	571 s	509.11 %	261 MB	83 %	9603 s	318.34 q/s	100 %
5	151.98 p/s	658 s	441.79 %	255 MB	81 %	9676 s	315.94 q/s	99 %
6	170.07 p/s	588 s	494.39 %	245 MB	78 %	9685 s	315.64 q/s	99 %

Table 5.8: AMD Athlon 5350 without Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	35.42 p/s	2823 s	-	317 MB	-	9600 s	318.44 q/s	-
2	68.82 p/s	1453 s	194.29 %	611 MB	193 %	10101 s	302.64 q/s	95 %
3	96.34 p/s	1038 s	271.97 %	742 MB	234 %	10971 s	278.64 q/s	88 %
4	134.41 p/s	744 s	379.44 %	878 MB	277 %	11519 s	265.39 q/s	83 %
5	110.74 p/s	903 s	312.62 %	996 MB	314 %	12129 s	252.04 q/s	79 %
6	118.20 p/s	846 s	333.69 %	1194 MB	377 %	12699 s	240.73 q/s	76 %

Table 5.9: Intel Pentium Dualcore E2140 with Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	56.34 p/s	1775 s	-	314 MB	-	5772 s	529.63 q/s	-
2	106.95 p/s	935 s	189.84 %	279 MB	89 %	5631 s	542.89 q/s	103 %
3	122.25 p/s	818 s	216.99 %	266 MB	85 %	5709 s	535.47 q/s	101 %
4	150.60 p/s	664 s	267.32 %	264 MB	84 %	5787 s	528.25 q/s	100 %

Table 5.10: Intel Pentium Dualcore E2140 without Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	56.21 p/s	1779 s	-	311 MB	-	5725 s	533.97 q/s	-
2	95.60 p/s	1046 s	170.08 %	601 MB	193 %	6099 s	501.23 q/s	94 %
3	95.24 p/s	1050 s	169.43 %	730 MB	235 %	6614 s	462.20 q/s	87 %
4	107.99 p/s	926 s	192.12 %	864 MB	278 %	7456 s	410.01 q/s	77 %

Table 5.11: Allwinner A20 SoC ARMv7-A, Banana Pro with Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	9.49 p/s	10539 s	-	61 MB	-	22453 s	136.15 q/s	-
2	19.60 p/s	5102 s	206.57 %	50 MB	82 %	23727 s	128.84 q/s	95 %
3	21.48 p/s	4655 s	226.40 %	54 MB	89 %	23960 s	127.59 q/s	94 %
4	23.07 p/s	4335 s	243.11 %	46 MB	75 %	24043 s	127.15 q/s	93 %

Table 5.12: Allwinner A20 SoC ARMv7-A, Banana Pro without Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	9.52 p/s	10500 s	-	73 MB	-	22499 s	135.87 q/s	-
2	19.14 p/s	5226 s	200.92 %	89 MB	122 %	25767 s	118.64 q/s	87 %
3	19.35 p/s	5167 s	203.21 %	101 MB	138 %	28917 s	105.72 q/s	78 %
4	19.79 p/s	5052 s	207.84 %	126 MB	173 %	31831 s	96.04 q/s	71 %

5.3.3 Profiling: Even more Speed with TCMalloc

Even though the multithreading performance is already better than expected, the greed for speed can be satisfied even more. Profiling code is often almost obligatory to eliminate hidden bottlenecks. The sample profiler[33] from Microsoft Visual Studio Enterprise 2015 revealed that over 55 % of the execution time was spent in system calls to malloc (33,85 %) and free (21,86 %). Many vector objects are filled and resized during the insertion process, which is the main reason why malloc and free are called this frequently. As a consequence, using a faster malloc/free implementation does have a big impact on the insertion speed. There are many malloc implementations which claim to be faster than the standard allocators operating systems provide. Thread-Caching Malloc (TCMalloc) from Google[34] is a popular one and it is also used with the old nanocube implementation[35]. In order to use TCMalloc, a program just needs to be linked with the library. The linker will "wire up" malloc and free calls into TCMalloc instead of the standard allocation library. Instructions can be found on the official GitHub Page[36].

The insertion speedup measured is on average 23,15 % (15 - 37 %). The queries got processed on average 13,47 % (10 – 19 %) faster. The memory consumption went up by an average of 2 %. Interestingly, the "additional" virtual (logical) cores of the Intel Core i7 CPU with Hyper-Threading are now used more efficiently, too. The kink in the line graph of the insertion speed (see fig. 5.5 Intel Core i7-4710HQ) shifted from seven to eight threads.

Table 5.13: TCMalloc Intel Core i7-4710HQ with Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	124.53 p/s	803 s	-	313 MB	-	3229 s	946.73 q/s	-
2	284.09 p/s	352 s	228.13 %	285 MB	91 %	3208 s	952.93 q/s	101 %
3	456.62 p/s	219 s	366.67 %	273 MB	87 %	3235 s	944.98 q/s	100 %
4	549.45 p/s	182 s	441.21 %	275 MB	88 %	3257 s	938.59 q/s	99 %
5	512.82 p/s	195 s	411.79 %	264 MB	84 %	3418 s	894.38 q/s	94 %
6	649.35 p/s	154 s	521.43 %	263 MB	84 %	3287 s	930.03 q/s	98 %
7	746.27 p/s	134 s	599.25 %	250 MB	80 %	3341 s	915.00 q/s	97 %
8	877.19 p/s	114 s	704.39 %	249 MB	80 %	3384 s	903.37 q/s	95 %
9	917.43 p/s	109 s	736.70 %	242 MB	77 %	3342 s	914.72 q/s	97 %
10	961.54 p/s	104 s	772.12 %	246 MB	79 %	3322 s	920.23 q/s	97 %

[33] "The sampling profiling method interrupts the computer processor at set intervals and collects the function call stack. Exclusive sample counts are incremented for the function that is executing and inclusive counts are incremented for all of the calling functions on the call stack. Sampling reports present the totals of these counts for the profiled module, function, source code line, and instruction." - https://msdn.microsoft.com/en-us/library/dd264994.aspx

[34] http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[35] https://github.com/laurolins/nanocube#thread-caching-malloc-tcmalloc

[36] https://github.com/gperftools/gperftools

Table 5.14: TCMalloc Intel Core i7-4710HQ without Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	127.23 p/s	786 s	-	311 MB	-	3233 s	945.56 q/s	-
2	253.16 p/s	395 s	198.99 %	604 MB	194 %	3441 s	888.40 q/s	94 %
3	364.96 p/s	274 s	286.86 %	735 MB	236 %	3636 s	840.76 q/s	89 %
4	469.48 p/s	213 s	369.01 %	875 MB	281 %	3831 s	797.96 q/s	84 %
5	476.19 p/s	210 s	374.29 %	1000 MB	322 %	3981 s	767.90 q/s	81 %
6	478.47 p/s	209 s	376.08 %	1195 MB	384 %	4145 s	737.52 q/s	78 %
7	467.29 p/s	214 s	367.29 %	1390 MB	447 %	4343 s	703.89 q/s	74 %
8	500.00 p/s	200 s	393.00 %	1478 MB	475 %	4489 s	681.00 q/s	72 %
9	497.51 p/s	201 s	391.04 %	1463 MB	470 %	4635 s	659.55 q/s	70 %
10	505.05 p/s	198 s	396.97 %	1598 MB	514 %	4767 s	641.28 q/s	68 %

Table 5.15: TCMalloc AMD Phenom II X4 955 Black Edition without Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	87.34 p/s	1145 s	-	309 MB	-	4302 s	710.60 q/s	-
2	194.17 p/s	515 s	222.33 %	279 MB	90 %	4370 s	699.54 q/s	98 %
3	322.58 p/s	310 s	369.35 %	270 MB	87 %	4376 s	698.58 q/s	98 %
4	448.43 p/s	223 s	513.45 %	264 MB	85 %	4358 s	701.47 q/s	99 %
5	510.20 p/s	196 s	584.18 %	257 MB	83 %	4438 s	688.82 q/s	97 %
6	552.49 p/s	181 s	632.60 %	258 MB	83 %	4416 s	692.26 q/s	97 %

Table 5.16: TCMalloc AMD Phenom II X4 955 Black Edition without Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	85.84 p/s	1165 s	-	303 MB	-	4351 s	702.60 q/s	-
2	172.41 p/s	580 s	200.86 %	604 MB	199 %	4808 s	635.82 q/s	90 %
3	256.41 p/s	390 s	298.72 %	736 MB	243 %	4998 s	611.64 q/s	87 %
4	336.70 p/s	297 s	392.26 %	873 MB	288 %	5294 s	577.45 q/s	82 %
5	340.14 p/s	294 s	396.26 %	998 MB	329 %	5568 s	549.03 q/s	78 %
6	348.43 p/s	287 s	405.92 %	1194 MB	394 %	5836 s	523.82 q/s	75 %

Table 5.17: TCMalloc AMD Phenom X4 9550 with Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	48.22 p/s	2074 s	-	315 MB	-	7328 s	417.17 q/s	-
2	114.42 p/s	874 s	131.01 %	285 MB	92 %	7265 s	420.78 q/s	101 %
3	192.31 p/s	520 s	220.19 %	272 MB	88 %	7406 s	412.77 q/s	99 %
4	263.85 p/s	379 s	302.11 %	275 MB	89 %	7609 s	401.76 q/s	96 %
5	273.22 p/s	366 s	312.84 %	264 MB	85 %	7625 s	400.92 q/s	96 %
6	315.46 p/s	317 s	361.20 %	259 MB	84 %	7453 s	410.17 q/s	98 %

Table 5.18: TCMalloc AMD Phenom X4 9550 without Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	49.07 p/s	2038 s	-	309 MB	-	7359 s	415.41 q/s	-
2	101.11 p/s	989 s	117.80 %	604 MB	199 %	7953 s	384.38 q/s	93 %
3	149.70 p/s	668 s	174.40 %	739 MB	244 %	8781 s	348.14 q/s	84 %
4	198.41 p/s	504 s	231.15 %	861 MB	284 %	9656 s	316.59 q/s	76 %
5	173.31 p/s	577 s	201.91 %	1008 MB	333 %	10343 s	295.56 q/s	71 %
6	208.77 p/s	479 s	243.22 %	1204 MB	397 %	11328 s	269.86 q/s	65 %

Table 5.19: TCMalloc AMD Athlon 5350 with Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	41.44 p/s	2413 s	-	315 MB	-	8163 s	374.49 q/s	-
2	93.90 p/s	1065 s	226.57 %	285 MB	90 %	8233 s	371.31 q/s	99 %
3	151.98 p/s	658 s	366.72 %	271 MB	86 %	8330 s	366.99 q/s	98 %
4	204.92 p/s	488 s	494.47 %	280 MB	89 %	8493 s	359.94 q/s	96 %
5	210.08 p/s	476 s	506.93 %	264 MB	84 %	8393 s	364.23 q/s	97 %
6	248.76 p/s	402 s	600.25 %	269 MB	85 %	8417 s	363.19 q/s	97 %

Table 5.20: TCMalloc AMD Athlon 5350 without Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	41.61 p/s	2403 s	-	311 MB	-	8257 s	370.23 q/s	-
2	82.17 p/s	1217 s	231.96 %	606 MB	191 %	8951 s	341.53 q/s	107 %
3	124.22 p/s	805 s	350.68 %	738 MB	233 %	9473 s	322.71 q/s	101 %
4	163.13 p/s	613 s	460.52 %	874 MB	276 %	10062 s	303.82 q/s	95 %
5	138.12 p/s	724 s	389.92 %	1004 MB	317 %	10576 s	289.05 q/s	91 %
6	153.85 p/s	650 s	434.31 %	1203 MB	379 %	11238 s	272.02 q/s	85 %

Table 5.21: TCMalloc Intel Pentium Dualcore E2140 with Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	76.39 p/s	1309 s	-	313 MB	-	4914 s	622.10 q/s	-
2	170.65 p/s	586 s	223.38 %	285 MB	91 %	4976 s	614.35 q/s	99 %
3	188.32 p/s	531 s	246.52 %	276 MB	88 %	4976 s	614.35 q/s	99 %
4	207.04 p/s	483 s	271.01 %	263 MB	84 %	4992 s	612.38 q/s	98 %

Table 5.22: TCMalloc Intel Pentium Dualcore E2140 without Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	76.63 p/s	1305 s	-	309 MB	-	4898 s	624.13 q/s	-
2	147.71 p/s	677 s	192.76 %	603 MB	195 %	5319 s	574.73 q/s	92 %
3	137.36 p/s	728 s	179.26 %	743 MB	240 %	5616 s	544.34 q/s	87 %
4	151.98 p/s	658 s	198.33 %	873 MB	283 %	5974 s	511.72 q/s	82 %

Table 5.23: TCMalloc Allwinner A20 SoC ARMv7-A, Banana Pro with Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	11.47 p/s	8721 s	-	74 MB	-	19323 s	158.21 q/s	-
2	24.78 p/s	4035 s	216.13 %	57 MB	77 %	19891 s	153.69 q/s	97 %
3	27.23 p/s	3672 s	237.50 %	60 MB	81 %	19571 s	156.20 q/s	99 %
4	28.51 p/s	3508 s	248.60 %	53 MB	72 %	20332 s	150.35 q/s	95 %

Table 5.24: TCMalloc Allwinner A20 SoC ARMv7-A, Banana Pro without Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	11.41 p/s	8764 s	-	75 MB	-	19452 s	157.16 q/s	-
2	23.02 p/s	4344 s	201.75 %	86 MB	115 %	22485 s	135.96 q/s	87 %
3	23.26 p/s	4300 s	203.81 %	104 MB	139 %	24938 s	122.58 q/s	78 %
4	23.42 p/s	4269 s	205.29 %	128 MB	171 %	27565 s	110.90 q/s	71 %

Table 5.25: TCMalloc Broadcom BCM2835 SoC ARMv6, Raspberry Pi B rev. 2 with Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	6.22 p/s	16089 s	-	74 MB	-	38606 s	79.18 q/s	-
2	5.67 p/s	17645 s	91.18 %	57 MB	77 %	45169 s	67.68 q/s	85 %

Table 5.26: TCMalloc Broadcom BCM2835 SoC ARMv6, Raspberry Pi B rev. 2 without Quadtree Partition

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	6.32 p/s	15828 s	-	78 MB	-	35653 s	85.74 q/s	-
2	5.91 p/s	16909 s	93.61 %	98 MB	126 %	44195 s	69.17 q/s	81 %

Table 5.27: TCMalloc Intel Core i7-4710HQ with Quadtree Partition: up to 100 Threads

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
1	131.75 p/s	759 s	-	315 MB	-	3253 s	939.75 q/s	-
2	281.69 p/s	355 s	213.80 %	287 MB	91 %	3261 s	937.44 q/s	100 %
3	446.43 p/s	224 s	338.84 %	273 MB	87 %	3221 s	949.08 q/s	101 %
4	574.71 p/s	174 s	436.21 %	286 MB	91 %	3211 s	952.04 q/s	101 %
5	671.14 p/s	149 s	509.40 %	258 MB	82 %	3251 s	940.33 q/s	100 %
6	775.19 p/s	129 s	588.37 %	265 MB	84 %	3271 s	934.58 q/s	99 %
7	833.33 p/s	120 s	632.50 %	255 MB	81 %	3273 s	934.01 q/s	99 %
8	900.90 p/s	111 s	683.78 %	250 MB	79 %	3269 s	935.15 q/s	100 %
9	934.58 p/s	107 s	709.35 %	242 MB	77 %	3290 s	929.18 q/s	99 %
10	1010.10 p/s	99 s	766.67 %	252 MB	80 %	3330 s	918.02 q/s	98 %
11	1041.67 p/s	96 s	790.63 %	239 MB	76 %	3301 s	926.08 q/s	99 %
12	1111.11 p/s	90 s	843.33 %	240 MB	76 %	3369 s	907.39 q/s	97 %
13	1136.36 p/s	88 s	862.50 %	230 MB	73 %	3341 s	915.00 q/s	97 %
14	1176.47 p/s	85 s	892.94 %	237 MB	75 %	3362 s	909.28 q/s	97 %
15	1234.57 p/s	81 s	937.04 %	239 MB	76 %	3390 s	901.77 q/s	96 %
16	1234.57 p/s	81 s	937.04 %	242 MB	77 %	3415 s	895.17 q/s	95 %
17	1315.79 p/s	76 s	998.68 %	234 MB	74 %	3382 s	903.90 q/s	96 %
18	1369.86 p/s	73 s	1039.73 %	231 MB	73 %	3431 s	890.99 q/s	95 %
19	1408.45 p/s	71 s	1069.01 %	234 MB	74 %	3453 s	885.32 q/s	94 %
20	1428.57 p/s	70 s	1084.29 %	232 MB	74 %	3444 s	887.63 q/s	94 %
21	1470.59 p/s	68 s	1116.18 %	238 MB	76 %	3443 s	887.89 q/s	94 %
22	1587.30 p/s	63 s	1204.76 %	227 MB	72 %	3456 s	884.55 q/s	94 %
23	1562.50 p/s	64 s	1185.94 %	234 MB	74 %	3462 s	883.02 q/s	94 %
24	1612.90 p/s	62 s	1224.19 %	227 MB	72 %	3481 s	878.20 q/s	93 %
25	1612.90 p/s	62 s	1224.19 %	232 MB	74 %	3530 s	866.01 q/s	92 %
26	1785.71 p/s	56 s	1355.36 %	226 MB	72 %	3521 s	868.22 q/s	92 %
27	1754.39 p/s	57 s	1331.58 %	229 MB	73 %	3554 s	860.16 q/s	92 %
28	1818.18 p/s	55 s	1380.00 %	229 MB	73 %	3552 s	860.64 q/s	92 %
29	1818.18 p/s	55 s	1380.00 %	229 MB	73 %	3552 s	860.64 q/s	92 %
30	1886.79 p/s	53 s	1432.08 %	228 MB	72 %	3564 s	857.74 q/s	91 %
31	1923.08 p/s	52 s	1459.62 %	227 MB	72 %	3565 s	857.50 q/s	91 %

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
32	1960.78 p/s	51 s	1488.24 %	228 MB	72 %	3583 s	853.20 q/s	91 %
33	2000.00 p/s	50 s	1518.00 %	223 MB	71 %	3613 s	846.11 q/s	90 %
34	2000.00 p/s	50 s	1518.00 %	225 MB	71 %	3589 s	851.77 q/s	91 %
35	2083.33 p/s	48 s	1581.25 %	226 MB	72 %	3640 s	839.84 q/s	89 %
36	2083.33 p/s	48 s	1581.25 %	224 MB	71 %	3667 s	833.65 q/s	89 %
37	2127.66 p/s	47 s	1614.89 %	225 MB	71 %	3663 s	834.56 q/s	89 %
38	2222.22 p/s	45 s	1686.67 %	219 MB	70 %	3642 s	839.37 q/s	89 %
39	2272.73 p/s	44 s	1725.00 %	225 MB	71 %	3680 s	830.71 q/s	88 %
40	2272.73 p/s	44 s	1725.00 %	219 MB	70 %	3690 s	828.46 q/s	88 %
41	2325.58 p/s	43 s	1765.12 %	219 MB	70 %	3662 s	834.79 q/s	89 %
42	2380.95 p/s	42 s	1807.14 %	221 MB	70 %	3713 s	823.32 q/s	88 %
43	2380.95 p/s	42 s	1807.14 %	221 MB	70 %	3720 s	821.77 q/s	87 %
44	2439.02 p/s	41 s	1851.22 %	225 MB	71 %	3739 s	817.60 q/s	87 %
45	2500.00 p/s	40 s	1897.50 %	220 MB	70 %	3738 s	817.82 q/s	87 %
46	2500.00 p/s	40 s	1897.50 %	216 MB	69 %	3766 s	811.74 q/s	86 %
47	2564.10 p/s	39 s	1946.15 %	223 MB	71 %	3807 s	802.99 q/s	85 %
48	2564.10 p/s	39 s	1946.15 %	219 MB	70 %	3771 s	810.66 q/s	86 %
49	2631.58 p/s	38 s	1997.37 %	220 MB	70 %	3844 s	795.27 q/s	85 %
50	2702.70 p/s	37 s	2051.35 %	220 MB	70 %	3812 s	801.94 q/s	85 %
51	2631.58 p/s	38 s	1997.37 %	221 MB	70 %	3979 s	768.28 q/s	82 %
52	2702.70 p/s	37 s	2051.35 %	224 MB	71 %	3905 s	782.84 q/s	83 %
53	2777.78 p/s	36 s	2108.33 %	220 MB	70 %	3863 s	791.35 q/s	84 %
54	2777.78 p/s	36 s	2108.33 %	221 MB	70 %	3892 s	785.46 q/s	84 %
55	2941.18 p/s	34 s	2232.35 %	217 MB	69 %	3906 s	782.64 q/s	83 %
56	2777.78 p/s	36 s	2108.33 %	219 MB	70 %	3937 s	776.48 q/s	83 %
57	2857.14 p/s	35 s	2168.57 %	222 MB	70 %	3891 s	785.66 q/s	84 %
58	2941.18 p/s	34 s	2232.35 %	224 MB	71 %	3941 s	775.69 q/s	83 %
59	3125.00 p/s	32 s	2371.88 %	215 MB	68 %	3941 s	775.69 q/s	83 %
60	3030.30 p/s	33 s	2300.00 %	220 MB	70 %	3947 s	774.51 q/s	82 %
61	3030.30 p/s	33 s	2300.00 %	221 MB	70 %	3964 s	771.19 q/s	82 %
62	3125.00 p/s	32 s	2371.88 %	221 MB	70 %	3966 s	770.80 q/s	82 %
63	3125.00 p/s	32 s	2371.88 %	219 MB	70 %	3966 s	770.80 q/s	82 %
64	3225.81 p/s	31 s	2448.39 %	223 MB	71 %	3995 s	765.21 q/s	81 %
65	3225.81 p/s	31 s	2448.39 %	222 MB	70 %	4006 s	763.11 q/s	81 %
66	3225.81 p/s	31 s	2448.39 %	226 MB	72 %	4003 s	763.68 q/s	81 %
67	3333.33 p/s	30 s	2530.00 %	223 MB	71 %	4016 s	761.21 q/s	81 %
68	3333.33 p/s	30 s	2530.00 %	225 MB	71 %	4077 s	749.82 q/s	80 %
69	3333.33 p/s	30 s	2530.00 %	219 MB	70 %	4080 s	749.26 q/s	80 %
70	3448.28 p/s	29 s	2617.24 %	221 MB	70 %	4071 s	750.92 q/s	80 %
71	3333.33 p/s	30 s	2530.00 %	225 MB	71 %	4091 s	747.25 q/s	80 %
72	3448.28 p/s	29 s	2617.24 %	219 MB	70 %	4056 s	753.70 q/s	80 %
73	3448.28 p/s	29 s	2617.24 %	220 MB	70 %	4121 s	741.81 q/s	79 %

T.	Speed	Time	Speedup	Memory	rel.	Query time	Query speed	rel.
74	3448.28 p/s	29 s	2617.24 %	227 MB	72 %	4138 s	738.76 q/s	79 %
75	3571.43 p/s	28 s	2710.71 %	221 MB	70 %	4122 s	741.63 q/s	79 %
76	3703.70 p/s	27 s	2811.11 %	223 MB	71 %	4129 s	740.37 q/s	79 %
77	3571.43 p/s	28 s	2710.71 %	225 MB	71 %	4151 s	736.45 q/s	78 %
78	3703.70 p/s	27 s	2811.11 %	227 MB	72 %	4167 s	733.62 q/s	78 %
79	3703.70 p/s	27 s	2811.11 %	228 MB	72 %	4197 s	728.38 q/s	78 %
80	3703.70 p/s	27 s	2811.11 %	227 MB	72 %	4158 s	735.21 q/s	78 %
81	3703.70 p/s	27 s	2811.11 %	228 MB	72 %	4196 s	728.55 q/s	78 %
82	3846.15 p/s	26 s	2919.23 %	227 MB	72 %	4178 s	731.69 q/s	78 %
83	3846.15 p/s	26 s	2919.23 %	228 MB	72 %	4224 s	723.72 q/s	77 %
84	3846.15 p/s	26 s	2919.23 %	227 MB	72 %	4268 s	716.26 q/s	76 %
85	4000.00 p/s	25 s	3036.00 %	227 MB	72 %	4277 s	714.75 q/s	76 %
86	4000.00 p/s	25 s	3036.00 %	231 MB	73 %	4243 s	720.48 q/s	77 %
87	3846.15 p/s	26 s	2919.23 %	233 MB	74 %	4270 s	715.93 q/s	76 %
88	4000.00 p/s	25 s	3036.00 %	231 MB	73 %	4275 s	715.09 q/s	76 %
89	4166.67 p/s	24 s	3162.50 %	233 MB	74 %	4363 s	700.66 q/s	75 %
90	4000.00 p/s	25 s	3036.00 %	236 MB	75 %	4321 s	707.48 q/s	75 %
91	3846.15 p/s	26 s	2919.23 %	234 MB	74 %	4323 s	707.15 q/s	75 %
92	4166.67 p/s	24 s	3162.50 %	236 MB	75 %	4337 s	704.87 q/s	75 %
93	4166.67 p/s	24 s	3162.50 %	237 MB	75 %	4347 s	703.24 q/s	75 %
94	4166.67 p/s	24 s	3162.50 %	239 MB	76 %	4346 s	703.41 q/s	75 %
95	4166.67 p/s	24 s	3162.50 %	234 MB	74 %	4360 s	701.15 q/s	75 %
96	4166.67 p/s	24 s	3162.50 %	237 MB	75 %	4401 s	694.61 q/s	74 %
97	4166.67 p/s	24 s	3162.50 %	236 MB	75 %	4387 s	696.83 q/s	74 %
98	4166.67 p/s	24 s	3162.50 %	240 MB	76 %	4412 s	692.88 q/s	74 %
99	4545.45 p/s	22 s	3450.00 %	239 MB	76 %	4420 s	691.63 q/s	74 %
100	4347.83 p/s	23 s	3300.00 %	238 MB	76 %	4437 s	688.98 q/s	73 %

6 Save and Load

This chapter describes how the in-memory data structure Nanocubes is saved to disk and loaded back into memory.

The data structure mostly consists of intertwined 64-bit pointers in RAM. Normal pointers store absolute addresses in the virtual address space of a program. When loading back a nanocube from disk into memory, the objects the pointers pointed to are located at random addresses on potentially every program start. This is due to the Address Space Layout Randomization (ASLR), which practically every modern PC operating system performs. Storing absolute addresses does imply the need to traverse the whole data structure to make up for the random address offset the operating system applied to the nanocube process. This tedious procedure can be circumvented by storing relative addresses. *Offset pointers* store the distance from their own *this* pointer to the object they point to. As a consequence, calculating the conversion from and to absolute addresses is an overhead of subtracting resp. adding two 64-bit integers. "Dereferencing" an offset pointer yields the correct absolute address even if the offset pointer and the referenced object are stored at addresses with a random offset applied from ASLR, because the stored distance is still the same inside the nanocube memory block.

Before the change to offset pointers, *tagged pointers* where used in several places like the container classes *small_raw_vector* and *small_vector* as well as in the *Node* and *Link* classes. Tagged pointers are pointers in which additional information is stored besides the actual address. In the case of Nanocubes, the most significant sixteen bits of the sixty-four bit pointers where used to store information like shared node flag, node type or last index number inserted into container. The offset in the offset pointers is stored in a sixty-four bit **signed integer**, which is why they cannot be tagged. It would mess up the two's complement, which is used to efficiently represent negative numbers in computers. Therefore, every information previously stored in tagged pointers is now stored in separate variables, resulting in a moderately higher memory usage.

The previous memory allocation strategy was composed of a pool and a Kernighan-Ritchie allocator, but was not effective. A *slab allocator* implementation from Lauro Lins, which uses offset pointers, is now in use. "With slab allocation, memory chunks suitable to fit data objects of certain type or size are preallocated. The slab allocator keeps track of these chunks, known as caches, so that when a request to allocate memory for a data object of a certain type is received, it can instantly satisfy the request with an already allocated slot. Destruction of the object does not free up the memory, but only opens a slot which is put in the list of free slots by the slab allocator. The next call to allocate memory of the same size will return the now unused memory slot. This process eliminates the need to search for suitable memory space and greatly alleviates memory fragmentation. In this context, a slab is one or more contiguous pages in the memory containing pre-allocated memory chunks."[1]

The slab allocator itself operates on a *memory-mapped file*. On Linux and Mac operating systems, the system call *mmap* is used to create such objects. On windows operating systems, the mmap wrapper *mman-win32*[2] is used to map the mmap function calls to the windows memory-map functions *CreateFileMapping* and *MapViewOfFile*. Read chapter 7 Nanocubes on Windows for more details on the differences between the operating systems regarding memory-mapped files.

A *MemoryBlock* object functions as an interface to the memory-mapped file, which the slab allocator uses to allocate memory itself in a sequential manner. In order to save a nanocube to disk, the memory block the nanocube is created on just needs to be written to a file. All necessary objects to reload a nanocube are located inside the *MemoryBlock*. Therefore, loading a nanocube is just a matter of linearly reading in the saved file back into a *MemoryBlock*. The allocator object instance can be "recovered" by simply assigning (casting) the start address (base pointer) of the *MemoryBlock* to an *Allocator* pointer, because the allocator stores itself as the first object in the *MemoryBlock*. This is similar to the *Nanocube* object(s), whose offset pointers are saved in the *_root* variable of an allocator object.

Due to the memory-mapping, nanocubes can be built, saved and loaded even if they exceed the size of physically available RAM. Classic hard disk drives are not suitable to compensate an insufficient amount of RAM, because the data structure nanocubes is not of a linear nature and therefore requires storage devices that are capable of performing fast random read and write operations. Solid-state drives (SSD), especially models with NVM Express (NVMe) interface, fulfill this requirement and could be used in production systems to work with very big nanocubes, that do not fit into RAM.

Every nanocube part has its own allocator object instance, because this enables the option to load different nanocubes (parts), that have the same nanocube scheme, as one nanocube. For the same reason, every nanocube part is saved into a separate file. This can be handy when working with historic data. For instance, nanocubes from last week can be loaded together with nanocubes from the current week to get a nanocube that visualizes both weeks' data.

6.1 Compression

A raw nanocube file is highly compressible. For example, the dataset mentioned in section 5.3 Benchmarks takes up over 17.2 GB as an eight parted nanocube. By using the bzip2 compression algorithm in standard mode, the files can be shrunk down to 1.93 GB (ca. 11 %).

To avoid additional steps when working with compressed nanocubes, direct gzip and bzip2 support is added. When loading nanocubes, bzip2 and gzip files are recognized by the file extensions `.bz2` and `.gz`. Enabling compression when storing nanocubes works alike.

Since the Boost C++ libraries[3] were already in use, Boost.Iostreams[4] was chosen to add

[1]https://en.wikipedia.org/wiki/Slab_allocation#Basis
[2]https://github.com/witwall/mman-win32
[3]http://www.boost.org/
[4]http://www.boost.org/doc/libs/1_61_0/libs/iostreams/doc/index.html

compression capabilities to the program. On windows operating systems gzip and bzip2 support is disabled by default. The library needs to be compiled with a command line supplying either file paths to the locations of the headers and binaries of the compression algorithms, if using pre-built binaries, or to the locations of the source files, if building from the source[5]. For example:

```
b2.exe link=static threading=multi address-model=64
-sBZIP2_SOURCE="E:\bzip2-1.0.6" -sZLIB_SOURCE="E:\zlib-1.2.8"
--build-type=complete stage
```

This will build Boost on windows as a static 64-bit multi-threading library with gzip and bzip2 support. The stated directories contain the source codes of the compression algorithms. This step might be necessary on other operating systems too, if pre-built libraries are in use[6].

To minimize the additional time spent when working with compression, nanocube parts get compressed and decompressed in parallel. The concurrent processing is implemented similar to the *Future* and *async* construct in the merging code described in section 5.2 C++11 Implementation.

A filtering stream buffer (*filtering_streambuf*) of type *input* from the boost iostreams library is used to decompress either gzip or bzip2 files on the fly into memory. A basic array sink of type *char* wraps the *MemoryBlock* a nanocube part is reloaded into. In order to use the *copy* method of the library, the basic array sink needs to be wrapped one more time into a *stream* object. *copy* reads from the filtering stream buffer, which performs the decompression, and writes it into the main memory resp. the chain of stream, sink and MemoryBlock.

Saving the data structure works very similar. A filtering stream buffer of the same type as above performs the compression on the fly while reading from a *array_source*, which wraps the MemoryBlock.

The nanocube schema[7] gets stored as a string with the name *annotation* in every *Nanocube* object (part). The annotation of the nanocube part, which got load first, is used to "reconstruct" the schema object. Given the vector of loaded nanocube objects and the schema, the *run* method inside the *main* function is called similarly to the procedure when building a new nanocube[8].

Listing 6.1: C++11 implementation for loading nanocubes back into memory

```
std::vector<std::string> fileNames;
boost::split(fileNames, options.load.getValue(),
                                        boost::is_any_of(","));

//read in every nanocube
std::vector<nanocube_type*> nanocubes;
std::mutex nanocubesMutex;
std::vector<std::future<nanocube_type*>> nanocubeFutures;
for (std::string fileName : fileNames) {
```

[5]http://www.boost.org/doc/libs/1_61_0/libs/iostreams/doc/installation.html

[6]Homebrew — The missing package manager for OS X, did install Boost without gzip and bzip2 support

[7]see section 4.1 Building a Nanocube from raw data

[8]read section 5.2 C++11 Implementation for more information

```
nanocubeFutures.push_back(std::async(std::launch::async,
    [&fileNames, &options, &nanocubes, &nanocubesMutex]
    (std::string fileName) {
    if (!fileExists(fileName)) {
        std::cout << "Can not load input file: " << fileName
                                                 << std::endl;
        std::flush(std::cout);
        throw std::runtime_error("Can not load input file");
    }
    else {
        //get filesize
        std::ifstream ifstream(fileName, std::ios::ate |
                                  std::ios::binary);
        auto fileSize = ifstream.tellg();
        ifstream.seekg(0, std::ios::beg);

        std::uint64_t arena_size =
            options.max_nanocube_size.isSet() ?
            ((uint64_t)options.max_nanocube_size.getValue()) *
            1024 * 1024 * 1024 :
            ARENA_SIZE;
        arena_size /= (std::uint64_t)fileNames.size();
#ifdef _WIN32
        std::wstring_convert<std::codecvt_utf8_utf16<wchar_t>>
                                                    converter;
        alloc::util::MMap* mmap = new alloc::util::MMap(
            arena_size, options.temp_path.isSet() ?
            &converter.from_bytes(options.temp_path.getValue()) :
                                                      nullptr);
#else
        alloc::util::MMap* mmap = new alloc::util::MMap(
                                                 arena_size);
#endif
        //read file into mmap
        auto mbBase = static_cast<char*>(
                                  mmap->memory_block().base());
        if (!boost::ends_with(fileName, ".gz") &&
            !boost::ends_with(fileName, ".bz2"))
            ifstream.read(mbBase, fileSize);
        else {
            boost::iostreams::filtering_streambuf<
                                  boost::iostreams::input> in;
            if (boost::ends_with(fileName, ".gz"))
                in.push(boost::iostreams::gzip_decompressor());
            else
                in.push(boost::iostreams::bzip2_decompressor());
```

```
                    in.push(ifstream);
                    boost::iostreams::basic_array_sink<char>
                                mbBaseArraySink(mbBase, arena_size);
                    boost::iostreams::stream<
                            boost::iostreams::basic_array_sink<char>>
                                 sMbBaseArraySink(mbBaseArraySink);
                    boost::iostreams::copy(in, sMbBaseArraySink);
                }

                //get SlabAllocator back from read in memoryblock
                Allocator* slab_allocator =
                                reinterpret_cast<Allocator*>(mbBase);
                SlabAllocatorWrapper::add(slab_allocator);

                //get nanocubes back from read in memory block
                return reinterpret_cast<nanocube_type*>(
                                                slab_allocator->root());
            }
        }, fileName));
}
//wait for threads to complete, store nanocube in vector
for (auto& nanocubeFuture : nanocubeFutures)
    nanocubes.push_back(nanocubeFuture.get());

auto firstNanocube = nanocubes.front();

// load header
std::cout << "Annotation: " << firstNanocube->getAnnotation()
                            << std::endl;
std::stringstream(firstNanocube->getAnnotation()) >> header;
std::cout << "Loaded nanocube from file" << std::endl;

Schema schema(header);

run(std::cin, nanocubes, schema);
```

Listing 6.2: C++11 implementation for storing nanocubes to disk

```
static void save(std::string fileName) {
    std::string fileEnding;
    std::string fileNameWithoutEnding;
    if (boost::ends_with(fileName, ".gz")) {
        fileEnding = ".nc.gz";
        fileNameWithoutEnding = fileName.substr(0, fileName.length()
                                                                 - 2);
    }
    else if (boost::ends_with(fileName, ".bz2")) {
        fileEnding = ".nc.bz2";
        fileNameWithoutEnding = fileName.substr(0, fileName.length()
                                                              - 3);
    }
    else {
        fileEnding = ".nc";
        fileNameWithoutEnding = fileName.substr(0, fileName.length()
                                                              - 2);
    }

    int fileNumber = 0;
    std::vector<std::future<void>> storFutures;
    for (Allocator* allocator : slab_allocators) {
        storFutures.push_back(std::async(std::launch::async,
                        [&](Allocator* allocator, int fileNumber) {
            std::string tmpFileName = fileNameWithoutEnding;
            std::ofstream ofstr(tmpFileName.append(
                    std::to_string(fileNumber).append(fileEnding)),
                    std::ios::binary);
            auto mb = allocator->memory_block();

            if (fileEnding == ".nc")
                ofstr.write(static_cast<const char*>(mb.base()),
                                                     mb.size());
            else {
                boost::iostreams::filtering_streambuf<
                                     boost::iostreams::input> in;
                if (fileEnding == ".nc.gz")
                    in.push(boost::iostreams::gzip_compressor());
                else//.bz2
                    in.push(boost::iostreams::bzip2_compressor());
                in.push(boost::iostreams::array_source(
                static_cast<const char*>(mb.base()), mb.size()));
                boost::iostreams::copy(in, ofstr);
            }
        }, allocator, fileNumber));

        fileNumber++;
    }

    //wait for threads to finish
    for (auto& storFuture : storFutures)
        storFuture.get();
}
```

6.2 32-bit Support

The nanocube implementation was intended to run only on 64-bit systems. The switch from tagged pointers to offset pointers has the fortunate side effect of making the program compatible to outdated but still used 32-bit computer systems. Even though the offset inside the offset pointers is stored as a 64-bit integer, dereferencing them on 32-bit systems works too, because truncating the first thirty-two most significant bits of a 64-bit pointer will never cause complications in this context; 32-bit operating systems return by definition only 32-bit memory addresses. Truncating tagged pointers would “cut off” the additional information stored inside the pointer, which would have caused problems.

The memory usage is significantly lower when using 32-bit binaries. For example, the same dataset used up about 40 % less RAM on the same computer when using a 32-bit compilation. This effect is probably due to the fact that 32-bit pointers are half the size of 64-bit pointers. I assume that modern C++ compilers are able to optimize out the obsolete part of the 64-bit (offset) pointers for 32-bit compilations. I can not think of another reason why a 32-bit nanocube would be about half the size of a 64-bit version. A noteworthy difference in build time was not encountered. Therefore, it does make sense to use 32-bit binaries even on 64-bit systems, if the build nanocube does not exceed the 32-bit memory boundary of 4 GB.

This boundary is also the reason why the maximum nanocube size must always be limited to at most 4 GB when using 32-bit binaries with the `-g` command line parameter in gigabytes. The default maximum nanocube size of 32 GB on windows and 1 TB on Linux and Mac cannot be addressed with a 32-bit process.

7 Nanocubes on Windows

This chapter describes the changes that were made to make the Nanocube project compatible with Microsoft Windows operating systems.

Nanocubes was primarily developed on Apple Mac OS with the integrated development environment (IDE) Xcode. Because C++11 is a very powerful and complex programming language, Xcode is up to this date[1] incapable of performing everyday refactoring task like renaming a variable or method. For that reason, I chose to continue programming with Microsoft Visual Studio 2015, which does support C++11 with proper refactoring.

Visual Studio resp. Windows operating systems do not use nor provide the *unistd.h* header file, which provides access to the POSIX operating system API on Unix-like systems[2]. On Windows, the header file *io.h* can be used instead. For that reason, every source file which includes *unistd.h* now uses a preprocessor condition to automatically include the correct header file depending on which operating system it is compiled on:

Listing 7.1: Precompiler directives to include header files depending on the operating system

```
#ifdef _WIN32
#include <io.h>
#else
#include <unistd.h>
#endif
```

The memory mapping implementations on windows and Linux/Mac behave differently. Linux and Mac use anonymous file mappings with a size of 1 TB, which are not backed by any file, accept the swap file if necessary. Creating them on Windows operating systems does require a contiguous memory block the size of the mapping. Due to the limited size and a possible fragmentation of the RAM and the swap file, a large enough contiguous memory blocks can be impossible to find. For that reason, the windows version of Nanocubes uses file-backed mappings instead. Temporary files are created, one for each nanocube part, with a total default size of 32 GB. The size can be adjusted with the `-g` command line parameter in gigabytes. Standardly, the files are located in the systems default temporary folder, but this can be changed with the `-w` command line parameter. This command line parameter is not present on compilations for Linux or Mac. The temporary files are deleted automatically by the operating system once all handles are closed, which is usually the case after exiting the nanocube program.

File streams and the standard input stream (stdin) operate in binary mode by default on Linux and Mac operation systems. Windows' default is text mode, which is why the *read*

[1]September 9, 2016

[2]https://en.wikipedia.org/wiki/Unistd.h

function in several places stopped reading at the first `0x0A` byte, which is the ASCII code for line feed indicating a newline. For that reason, every file stream constructor is now specifically parameterized to open the stream in binary mode. Moreover, on Windows the *_setmode* function is used to change the mode from standard input (stdin) to binary.

Visual Studio found programming errors which Xcode ignored like missing includes, variables which were out of scope and wrong typenames in template definitions. Visual Studio had a few smaller bugs too, which needed workarounds. In general, using both IDEs during the development of Multi-part Nanocubes was helpful, because often the same problem generated different error messages in the IDEs with a different degree of usefulness.

DMP files generated on windows use carriage return and line feed as line endings, instead of just a line feed. The nanocube code is adapted accordingly to support both types of line endings.

The third party libraries the nanocube program uses do all feature windows support.

8 Future

Further development of the Nanocube project is done by the Information Visualization department at AT&T Labs Research situated in New York City. Since the project is open source, others might participate, too. Currently[1], a compressed version of Nanocubes is in the works, which could greatly reduce the memory consumption by a factor of more than ten times. Faster queries will be possible, too. Moreover, faster insertion and the multi-part work described in this document will also apply naturally[2].

The quadtree partition resp. the split points of a nanocube part should be stored inside the nanocube file, too. This would remove the need to manually store the addresses, if adding new points later on is planned.

The prediction of the remaining build time could be improved by basing it on a conceived mathematical model that takes the rising insertion complexity into account.

[1]September 9, 2016

[2]Email from Lauro Lins to Lukas Scharlau, April 4, 2016

9 Appendix

9.1 Pseudocode: Merging two Nanocubes

The following pseudocode is based on the original nanocube pseudocode[1]. The function *Combine* recursively merges nanocube A into nanocube B. The resulting nanocube is equivalent to a nanocube built by the original pseudocode. Similarly to the nanocube paper[2], the code is designed for nanocubes with a temporal dimension as the last dimension.

Nanocube A is fully traversed in a depth-first search manner from left to right and compared against nanocube B to find and if necessary add missing data into it. The first function call is parameterized with the two root nodes of the nanocubes in question and the number of the first dimension level.

The code first recursively builds up a call stack down to the last node of the current dimension tree while creating missing nodes in nanocube B. Missing content nodes are created and shared nodes are copied to insert new additional content without corrupting the dataset. If the last dimension is reached all content from nanocube A's node is copied to the corresponding node in nanocube B. The copy function needs to only copy new data, that is not already present in nanocube B. When working the paths back from bottom to the top of the dimension trees, shared connections are made.

Notice that the original pseudo code in the nanocube paper has a bug that I found while working with it to develop this pseudo code:

"ShallowCopy as written in the pseudo-code only works for internal nodes, not for time series. Line 14 should be something like:[3]"

```
SetProperContent(node, d=dim(S) ?
    CopySummedTableTimeSeries(Content(node)) :
    ShallowCopy(Content(node)))
```

Listing 9.1: Pseudocode for merging two nanocubes

```
Combine(Node A, Node B, int dimension) {
    foreach(Childnode CA in A.childs) {
        Childnode CB = B.childs[CA.Lable];

        if(CB == null) {
            CB = new Node(CA.Lable);
            B.childs.add(CB);
```

[1][LKS13] Lins, Klosowski, and Scheidegger (2013): fig. 3, p. 3

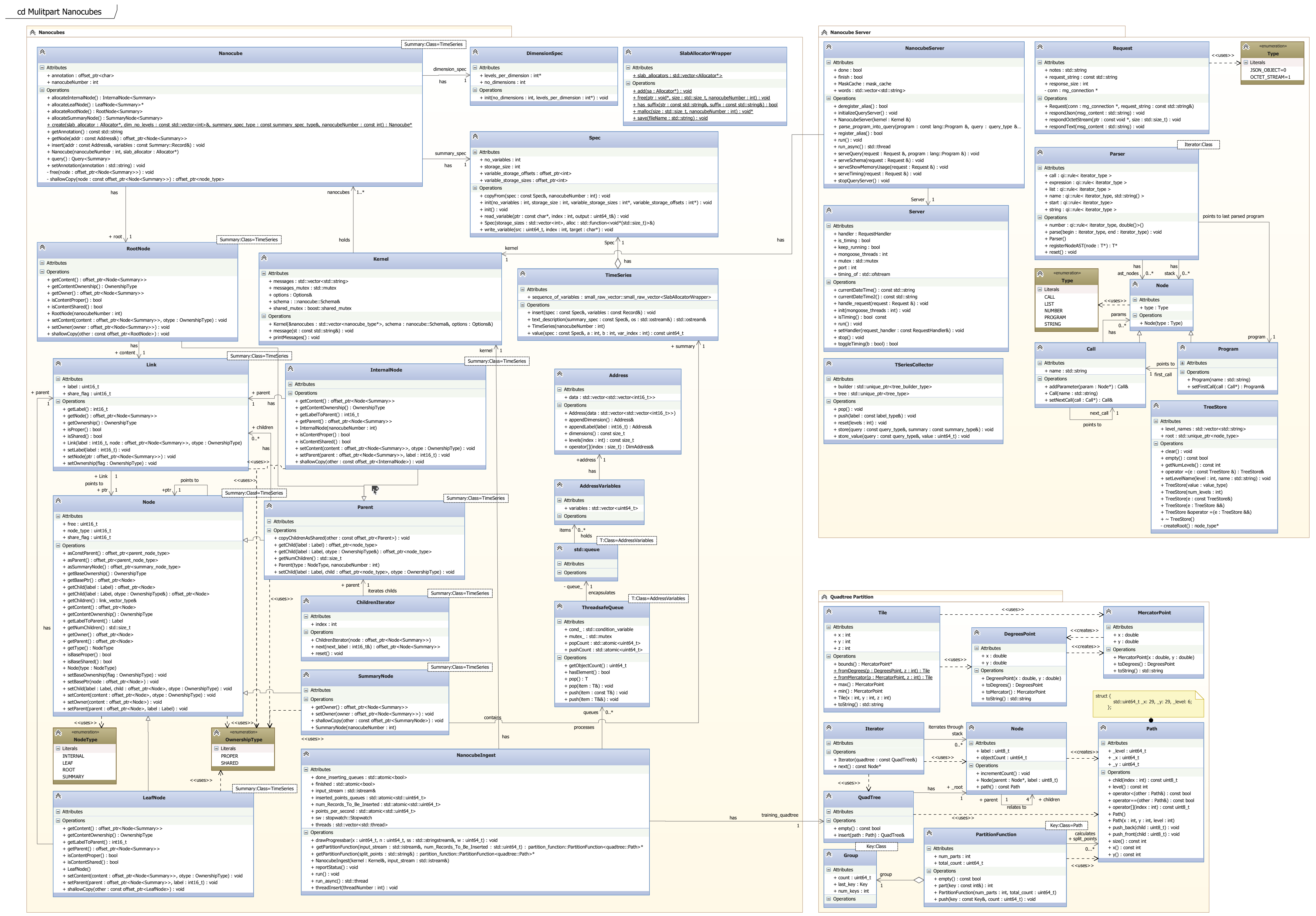
[2]cf. fn. footnote 1

[3]Lauro Lins: https://github.com/laurolins/nanocube/issues/31

```
            }
            //only in Flattree?
            else if(CB.isSharedChild) {
                CB = shallowCopy(CB);
                B.childs[CA.Lable] = CB;
            }

            Combine(CA, CB, dimension);
        }

        //Add Content from Node A to Node B

        bool isLastDimension = dimension == dim(S);

        if(B.childs.count == 1) {
            setSharedContent(B.Content, B.childs[0].Content);
            return; //nothing more to do for this node
        }
        else if(B.Content.isSharedContent)
            B.Content = isLastDimension ? copySummedTableTimeSeries(B.Content)
                                        : shallowCopy(B.Content);
        else if(B.Content == null)
            B.Content = isLastDimension ? new SummedTableTimeSeries()
                                        : new Node();

        if(isLastDimension) //Copy TableTimeSeries from Node A to Node B
            foreach(Object o in A.Content)
                B.Content.add(o); //Problem: Add only new objects!
        else //Combine the Content from Node A into Node B
            Combine(A.Content, B.Content, dimension + 1);
}
```

9.2 UML class diagram

The UML class diagram on the next page contains all classes and their relations mentioned in chapter 5 Multi-part Nanocubes. The diagram should help to get an overview of the program parts in question and the changes I made to implement Multi-part Nanocubes. The diagram does not include all classes nor all class relations of the nanocube program.


cd Mulitpart Nanocubes
Nanocubes
Summary:Class=TimeSeries
Nanocube
Attributes
+ annotation : offset_ptr<char>
+ nanocubeNumber : int
Operations
+ allocateInternalNode() : InternalNode<Summary>
+ allocateLeafNode() : LeafNode<Summary>*
+ allocateRootNode() : RootNode<Summary>
+ allocateSummaryNode() : SummaryNode<Summary>
+ create(slab_allocator : Allocator*, dim_no_levels : const std::vector<int>&, summary_spec_type : const summary_spec_type&, nanocubeNumber : const int) : Nanocube*
+ getAnnotation() : const std::string
+ getNode(addr : const Address&) : offset_ptr<Node<Summary>>
+ insert(addr : const Address&, variables : const Summary::Record&) : void
+ Nanocube(nanocubeNumber : int, slab_allocator : Allocator*)
+ query() : Query<Summary>
+ setAnnotation(annotation : std::string) : void
- free(node : offset_ptr<Node<Summary>>) : void
- shallowCopy(node : const offset_ptr<Node<Summary>>) : offset_ptr<node_type>
dimension_spec
has
1
summary_spec
has
1
DimensionSpec
Attributes
+ levels_per_dimension : int*
+ no_dimensions : int
Operations
+ init(no_dimensions : int, levels_per_dimension : int*) : void
SlabAllocatorWrapper
Attributes
+ slab_allocators : std::vector<Allocator*>
Operations
+ add(sa : Allocator*) : void
+ free(ptr : void*, size : std::size_t, nanocubeNumber : int) : void
+ has_suffix(str : const std::string&, suffix : const std::string&) : bool
+ malloc(size : std::size_t, nanocubeNumber : int) : void*
+ save(fileName : std::string) : void
Spec
Attributes
+ no_variables : int
+ storage_size : int
+ variable_storage_offsets : offset_ptr<int>
+ variable_storage_sizes : offset_ptr<int>
Operations
+ copyFrom(spec : const Spec&, nanocubeNumber : int) : void
+ init(no_variables : int, storage_size : int, variable_storage_sizes : int*, variable_storage_offsets : int*) : void
+ init() : void
+ read_variable(ptr : const char*, index : int, output : uint64_t&) : void
+ Spec(storage_sizes : std::vector<int>, alloc : std::function<void*(std::size_t)>&)
+ write_variable(src : uint64_t, index : int, target : char*) : void
has
+ root
1
nanocubes
1..*
holds
Spec
1
kernel
1
has
has
Summary:Class=TimeSeries
RootNode
Attributes
Operations
+ getContent() : offset_ptr<Node<Summary>>
+ getContentOwnership() : OwnershipType
+ getOwner() : offset_ptr<Node<Summary>>
+ isContentProper() : bool
+ isContentShared() : bool
+ RootNode(nanocubeNumber : int)
+ setContent(content : offset_ptr<Node<Summary>>, otype : OwnershipType) : void
+ setOwner(owner : offset_ptr<Node<Summary>>) : void
+ shallowCopy(other : const offset_ptr<RootNode>) : void
Kernel
Attributes
+ messages : std::vector<std::string>
+ messages_mutex : std::mutex
+ options : Options&
+ schema : ::nanocube::Schema&
+ shared_mutex : boost::shared_mutex
Operations
+ Kernel(&nanocubes : std::vector<nanocube_type*>, schema : nanocube::Schema&, options : Options&)
+ message(st : const std::string&) : void
+ printMessages() : void
TimeSeries
Attributes
+ sequence_of_variables : small_raw_vector::small_raw_vector<SlabAllocatorWrapper>
Operations
+ insert(spec : const Spec&, variables : const Record&) : void
+ text_description(summary_spec : const Spec&, os : std::ostream&) : std::ostream&
+ TimeSeries(nanocubeNumber : int)
+ value(spec : const Spec&, a : int, b : int, var_index : int) : const uint64_t
has
+ content
1
Summary:Class=TimeSeries
Link
Attributes
+ label : uint16_t
+ share_flag : uint16_t
Operations
+ getLabel() : int16_t
+ getNode() : offset_ptr<Node<Summary>>
+ getOwnership() : OwnershipType
+ isProper() : bool
+ isShared() : bool
+ Link(label : int16_t, node : offset_ptr<Node<Summary>>, otype : OwnershipType)
+ setLabel(label : int16_t) : void
+ setNode(ptr : offset_ptr<Node<Summary>>) : void
+ setOwnership(flag : OwnershipType) : void
+ parent
1
+ parent
1
has
+ children
0..*
has
<<uses>>
<<uses>>
kernel
1
Summary:Class=TimeSeries
InternalNode
Attributes
Operations
+ getContent() : offset_ptr<Node<Summary>>
+ getContentOwnership() : OwnershipType
+ getLabelToParent() : int16_t
+ getParent() : offset_ptr<Node<Summary>>
+ InternalNode(nanocubeNumber : int)
+ isContentProper() : bool
+ isContentShared() : bool
+ setContent(content : offset_ptr<Node<Summary>>, otype : OwnershipType) : void
+ setParent(parent : offset_ptr<Node<Summary>>, label : int16_t) : void
+ shallowCopy(other : const offset_ptr<InternalNode>) : void
+ summary
1
Address
Attributes
+ data : std::vector<std::vector<int16_t>>
Operations
+ Address(data : std::vector<std::vector<int16_t>>)
+ appendDimension() : Address&
+ appendLabel(label : int16_t) : Address&
+ dimensions() : const size_t
+ levels(index : int) : const size_t
+ operator[](index : size_t) : DimAddress&
+address
1
has
AddressVariables
Attributes
+ variables : std::vector<uint64_t>
Operations
items
0..*
holds
T:Class=AddressVariables
std::queue
Attributes
Operations
- queue_
1
encapsulates
T:Class=AddressVariables
ThreadsafeQueue
Attributes
+ cond_ : std::condition_variable
+ mutex_ : std::mutex
+ popCount : std::atomic<uint64_t>
+ pushCount : std::atomic<uint64_t>
Operations
+ getObjectCount() : uint64_t
+ hasElement() : bool
+ pop() : T
+ pop(item : T&) : void
+ push(item : const T&) : void
+ push(item : T&&) : void
queues
0..*
processes
+ Link
1
points to
+ ptr
1
points to
+ptr
1
Summary:Class=TimeSeries
Node
Attributes
+ free : uint16_t
+ node_type : uint16_t
+ share_flag : uint16_t
Operations
+ asConstParent() : offset_ptr<parent_node_type>
+ asParent() : offset_ptr<parent_node_type>
+ asSummaryNode() : offset_ptr<summary_node_type>
+ getBaseOwnership() : OwnershipType
+ getBasePtr() : offset_ptr<Node>
+ getChild(label : Label) : offset_ptr<Node>
+ getChild(label : Label, otype : OwnershipType&) : offset_ptr<Node>
+ getChildren() : link_vector_type&
+ getContent() : offset_ptr<Node>
+ getContentOwnership() : OwnershipType
+ getLabelToParent() : Label
+ getNumChildren() : std::size_t
+ getOwner() : offset_ptr<Node>
+ getParent() : offset_ptr<Node>
+ getType() : NodeType
+ isBaseProper() : bool
+ isBaseShared() : bool
+ Node(type : NodeType)
+ setBaseOwnership(flag : OwnershipType) : void
+ setBasePtr(node : offset_ptr<Node>) : void
+ setChild(label : Label, child : offset_ptr<Node>, otype : OwnershipType) : void
+ setContent(content : offset_ptr<Node>, otype : OwnershipType) : void
+ setOwner(content : offset_ptr<Node>) : void
+ setParent(parent : offset_ptr<Node>, label : Label) : void
has
Summary:Class=TimeSeries
Parent
Attributes
Operations
+ copyChildrenAsShared(other : const offset_ptr<Parent>) : void
+ getChild(label : Label) : offset_ptr<node_type>
+ getChild(label : Label, otype : OwnershipType&) : offset_ptr<node_type>
+ getNumChildren() : std::size_t
+ Parent(type : NodeType, nanocubeNumber : int)
+ setChild(label : Label, child : offset_ptr<node_type>, otype : OwnershipType) : void
<<uses>>
+ parent
1
iterates childs
Summary:Class=TimeSeries
ChildrenIterator
Attributes
+ index : int
Operations
+ ChildrenIterator(node : offset_ptr<Node<Summary>>)
+ next(next_label : int16_t&) : offset_ptr<Node<Summary>>
+ reset() : void
Summary:Class=TimeSeries
SummaryNode
Attributes
Operations
+ getOwner() : offset_ptr<Node<Summary>>
+ setOwner(owner : offset_ptr<Node<Summary>>) : void
+ shallowCopy(other : const offset_ptr<SummaryNode>) : void
+ SummaryNode(nanocubeNumber : int)
contains
has
<<uses>>
<<uses>>
<<uses>>
«enumeration»
NodeType
Literals
INTERNAL
LEAF
ROOT
SUMMARY
«enumeration»
OwnershipType
Literals
PROPER
SHARED
<<uses>>
Summary:Class=TimeSeries
LeafNode
Attributes
Operations
+ getContent() : offset_ptr<Node<Summary>>
+ getContentOwnership() : OwnershipType
+ getLabelToParent() : int16_t
+ getParent() : offset_ptr<Node<Summary>>
+ isContentProper() : bool
+ isContentShared() : bool
+ LeafNode()
+ setContent(content : offset_ptr<Node<Summary>>, otype : OwnershipType) : void
+ setParent(parent : offset_ptr<Node<Summary>>, label : int16_t) : void
+ shallowCopy(other : const offset_ptr<LeafNode>) : void
NanocubeIngest
Attributes
+ done_inserting_queues : std::atomic<bool>
+ finished : std::atomic<bool>
+ input_stream : std::istream&
+ inserted_points_queues : std::atomic<std::uint64_t>
+ num_Records_To_Be_Inserted : std::atomic<std::uint64_t>
+ points_per_second : std::atomic<std::uint64_t>
+ sw : stopwatch::Stopwatch
+ threads : std::vector<std::thread>
Operations
+ drawProgressbar(x : uint64_t, n : uint64_t, ss : std::stringstream&, w : uint64_t) : void
+ getPartitionFunction(input_stream : std::istream&, num_Records_To_Be_Inserted : std::uint64_t) : partition_function::PartitionFunction<quadtree::Path>*
+ getPartitionFunction(split_points : std::string&) : partition_function::PartitionFunction<quadtree::Path>*
+ NanocubeIngest(kernel : Kernel&, input_stream : std::istream&)
+ reportStatus() : void
+ run() : void
+ run_async() : std::thread
+ threadInsert(threadNumber : int) : void
has
training_quadtree
1
Nanocube Server
NanocubeServer
Attributes
+ done : bool
+ finish : bool
+ MaskCache : mask_cache
+ words : std::vector<std::string>
Operations
+ deregister_alias() : bool
+ initializeQueryServer() : void
+ NanocubeServer(kernel : Kernel &)
+ parse_program_into_query(program : const lang::Program &, query : query_type &...
+ register_alias() : bool
+ run() : void
+ run_async() : std::thread
+ serveQuery(request : Request &, program : lang::Program &) : void
+ serveSchema(request : Request &) : void
+ serveShowMemoryUsage(request : Request &) : void
+ serveTiming(request : Request &) : void
+ stopQueryServer() : void
Server
1
Request
Attributes
+ notes : std::string
+ request_string : const std::string
+ response_size : int
- conn : mg_connection *
Operations
+ Request(conn : mg_connection *, request_string : const std::string&)
+ respondJson(msg_content : std::string) : void
+ respondOctetStream(ptr : const void *, size : std::size_t) : void
+ respondText(msg_content : std::string) : void
<<uses>>
«enumeration»
Type
Literals
JSON_OBJECT=0
OCTET_STREAM=1
Iterator:Class
Parser
Attributes
+ call : qi::rule< iterator_type >
+ expression : qi::rule< iterator_type >
+ list : qi::rule< iterator_type >
+ name : qi::rule< iterator_type, std::string() >
+ start : qi::rule< iterator_type>
+ string : qi::rule< iterator_type >
Operations
+ number : qi::rule< iterator_type, double()>()
+ parse(begin : iterator_type, end : iterator_type) : void
+ Parser()
+ registerNodeAST(node : T*) : T*
+ reset() : void
points to last parsed program
Server
Attributes
+ handler : RequestHandler
+ is_timing : bool
+ keep_running : bool
+ mongoose_threads : int
+ mutex : std::mutex
+ port : int
+ timing_of : std::ofstream
Operations
+ currentDateTime() : const std::string
+ currentDateTime2() : const std::string
+ handle_request(request : Request &) : void
+ init(mongoose_threads : int) : void
+ isTiming() : bool const
+ run() : void
+ setHandler(request_handler : const RequestHandler&) : void
+ stop() : void
+ toggleTiming(b : bool) : bool
has
ast_nodes
0..*
has
stack
0..*
«enumeration»
Type
Literals
CALL
LIST
NUMBER
PROGRAM
STRING
<<uses>>
Node
Attributes
+ type : Type
Operations
+ Node(type : Type)
params
0..*
has
program
1
TSeriesCollector
Attributes
+ builder : std::unique_ptr<tree_builder_type>
+ tree : std::unique_ptr<tree_type>
Operations
+ pop() : void
+ push(label : const label_type&) : void
+ reset(levels : int) : void
+ store(query : const query_type&, summary : const summary_type&) : void
+ store_value(query : const query_type&, value : uint64_t) : void
Call
Attributes
+ name : std::string
Operations
+ addParameter(param : Node*) : Call&
+ Call(name : std::string)
+ setNextCall(call : Call*) : Call&
points to
1
first_call
Program
Attributes
Operations
+ Program(name : std::string)
+ setFirstCall(call : Call*) : Program&
next_call
1
points to
TreeStore
Attributes
+ level_names : std::vector<std::string>
+ root : std::unique_ptr<node_type>
Operations
+ clear() : void
+ empty() : const bool
+ getNumLevels() : const int
+ operator =(e : const TreeStore &) : TreeStore&
+ setLevelName(level : int, name : std::string) : void
+ TreeStore(value : value_type)
+ TreeStore(num_levels : int)
+ TreeStore(e : const TreeStore&)
+ TreeStore(e : TreeStore &&)
+ TreeStore &operator =(e : TreeStore &&)
+ ~ TreeStore()
- createRoot() : node_type*
Quadtree Partition
Tile
Attributes
+ x : int
+ y : int
+ z : int
Operations
+ bounds() : MercatorPoint*
+ fromDegrees(p : DegreesPoint, z : int) : Tile
+ fromMercator(p : MercatorPoint, z : int) : Tile
+ max() : MercatorPoint
+ min() : MercatorPoint
+ Tile(x : int, y : int, z : int)
+ toString() : std::string
<<uses>>
<<uses>>
DegreesPoint
Attributes
+ x : double
+ y : double
Operations
+ DegreesPoint(x : double, y : double)
+ toDegrees() : DegreesPoint
+ toMercator() : MercatorPoint
+ toString() : std::string
<<creates>>
<<creates>>
MercatorPoint
Attributes
+ x : double
+ y : double
Operations
+ MercatorPoint(x : double, y : double)
+ toDegrees() : DegreesPoint
+ toString() : std::string
struct {
std::uint64_t _x: 29, _y: 29, _level: 6;
};
Iterator
Attributes
Operations
+ Iterator(quadtree : const QuadTree&)
+ next() : const Node*
iterrates through
stack
0..*
<<uses>>
Node
Attributes
+ label : uint8_t
+ objectCount : uint64_t
Operations
+ incrementCount() : void
+ Node(parent : Node*, label : uint8_t)
+ path() : const Path
<<creates>>
Path
Attributes
+ _level : uint64_t
+ _x : uint64_t
+ _y : uint64_t
Operations
+ child(index : int) : const uint8_t
+ level() : const int
+ operator<(other : Path&) : const bool
+ operator==(other : Path&) : const bool
+ operator[](index : int) : const uint8_t
+ Path()
+ Path(x : int, y : int, level : int)
+ push_back(child : uint8_t) : void
+ push_front(child : uint8_t) : void
+ size() : const int
+ x() : const int
+ y() : const int
<<uses>>
has
+ _root
1
+ parent
1
4
+ children
relates to
QuadTree
Attributes
Operations
+ empty() : const bool
+ insert(path : Path) : QuadTree&
<<uses>>
Key:Class
Key:Class=Path
PartitionFunction
Attributes
+ num_parts : int
+ total_count : uint64_t
Operations
+ empty() : const bool
+ part(key : const int&) : int
+ PartitionFunction(num_parts : int, total_count : uint64_t)
+ push(key : const Key&, count : uint64_t) : void
calculates
+ split_points
0...*
<<uses>>
Group
Attributes
+ count : uint64_t
+ last_key : Key
+ num_keys : int
Operations
group
1

Bibliography

[LKS13] Lauro Lins, James Klosowski, and Carlos Scheidegger. *Nanocubes for Real-Time Exploration of Spatiotemporal Datasets*. Tech. rep. IEEE InfoVis, Oct. 2013. URL: http://web1.research.att.com:81/techdocs_downloads/TD:101151_DS1_2013-10-02T14:31:49.560Z.pdf.