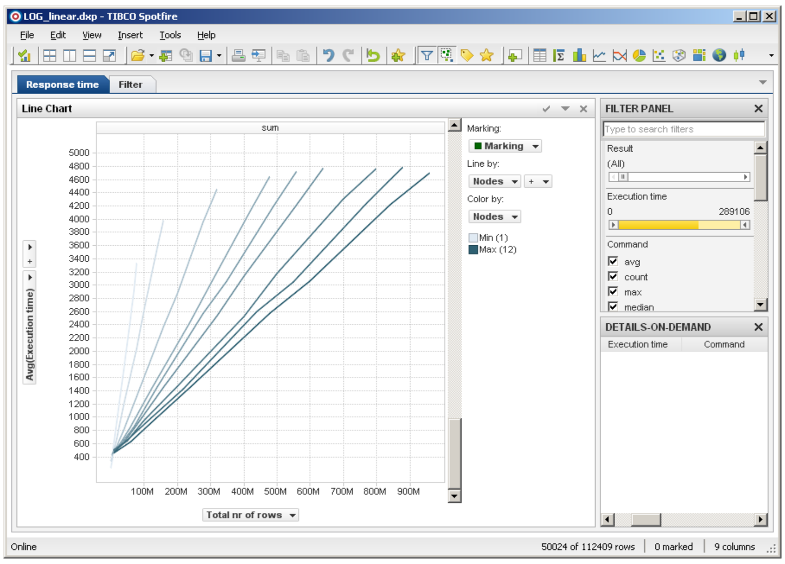

Distributed computations

As a back-end to a visualization tool

Master of Science Thesis
Foundations of Computations: Algorithms and Logic
Secure and Dependable Computer Systems

Felix Ekblom
Per Stolpe

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, June 2010

Distributed Computations
As a back-end to a visualization tool

Felix Ekblom
Per Stolpe

Examiner: Marina Papatriantafilou

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
The TIBCO Spotfire visualization client

Department of Computer Science and Engineering
Göteborg, Sweden June 2010

Abstract

Data sets used in analysis is rapidly outgrowing the size of memory used in the typical visualization tool. In order to circumvent this, a distributed system approach for the data processing part of the visualization tool is investigated. Response time is consider the most important performance metric. A distributed system approach is shown to be able to give reasonable response times and scales well with input data size.

Contents

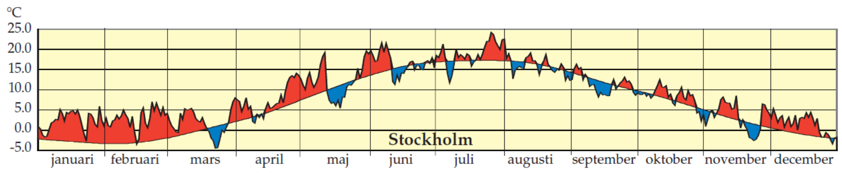

Figure 1: A diagram describing temperature change over the year [29]

1 Introduction

1.1 Background

In the past couple of decades, companies have gone from using folders with papers and registers on their business to keeping the data digitalized within computers. It is now possible to keep track of inventories, orders, customer history and much more in near real-time on a computer.

Many companies also save all their collected data, from for example sales, and try to use it to gain a business advantage by finding patterns or predicting results. By using this information, the companies can for example create personalized commercials or learn about temporal habits of their customers.

The process of examining the data is called data analysis, and there are several ways to perform the analysis. One way of doing it is by letting a human explore the data, looking for interesting patterns or testing out ideas. One of the human analyst's tools is the visualization application, a program that can present the data using visual aiding features, such as for example colors and diagrams. The main idea behind using a visualization tool is that it is easier for an analyst to understand the data when (s)he can see how it looks like. The common term for using analysis to gain advantages is business intelligence.

Visualization of data means displaying the data set visually. See Figure 1 for an example where the average temperature per day over the year is visualized as an area chart. Visualization is based around querying the data, which could be considered read-only, for properties and then creating a visualization of this to the analyst. Consider again Figure 1, the query behind the visualization might have been something like "Give me the average temperature per day for all days in the year 2009, order the result in the order the days in increasing order by date". By making this distinction between the data queries and the visualization of the result, one can focus on making the data query-part able to handle all kinds of data and return the result according to an agreed upon protocol to the visualization tool, which then visualizes the result.

For a visualization tool to be useful, given that a human has to constantly make decisions of what to do next, they need to be fast. The analyst must get feedback within seconds [27] to be able to maintain his chain of thoughts, which means that the data querying has to be very fast in order for the whole process to be fast enough.

Big companies can collect lots of data in a single day, where data sizes in the order of gigabytes are not unusual. Wal-mart is famous for its history of saving

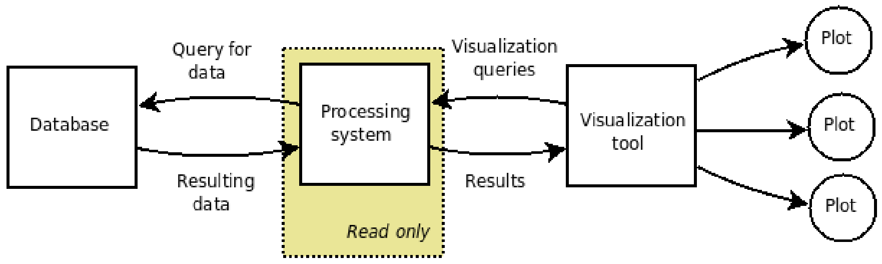

Figure 2: A structural decription of a visualization tool

customer data and sales data in big data warehouses [17]. With this size of data nowadays growing into petabytes, analysis gets much more complicated. For starters, the data might not even fit on the disks of a single computer, but may be spread out across a number of computers. Because of the growing gap between CPU and disk speed, one would be limited to the read throughput on the disks of a computer if a single computer is used.

A visualization tool might have a structure similar to Figure 2, where the system has 3 logical sub parts. Part 1 represents the data to be analyzed, stored in some kind of data storage, where storage system ranges from databases to text files. Subsystem 2 and 3 combined is the actual visualization tool, with the natural splitting of a local query system, performing fast responses to queries, and one part focused at creating visualizations based on some queries and their result.

The problem with a visualization tool implementing step 2 in Figure 2 locally is that the size of the data that the computer can handle efficiently is rather limited. Using the disk as primary data storage is generally not fast enough is a desktop environment, which makes the size of RAM the limiting factor.

1.2 Short problem statement

As the size of the data grows, a single computer is not enough for processing and visualizing the data while still obeying the demand of a fast response time. Running a visualization tool on a single desktop computer, to stay within the deadlines of a few seconds for a query, the data, or at least an index of the data, may need to reside within the memory of the computer.

This report will deal with the problems of building a system for handling data querying involving simple processing on read-only data in a distributed manner. are could correspond to using several computers for doing the computations performed by sub system 2 in Figure 2.

There are several existing distributed systems for computing over big data sets. An interesting approach is the Map/Reduce abstraction, which has an open source implementation in Hadoop. Hadoop has proved itself very capable in for example the field of web search engines, where it is used to build up indexes, and should be investigated as a back-end for the visualization tool queries.

1.3 Goal

Interesting questions which we aim to answer include how a system for distributed calculations on a data set could be built, how the data could be partitioned on the nodes in the system, and what response times one is to expect.

This report investigates and prototypes how distributed computations can be used to scale common operations performed in a data analysis application, such as for example statistical aggregations like summing and calculating the average value of a set. The main goal with the prototype is not to have a finished working product, but rather to have a working prototype and use it to gain performance measurements. With help from studies and our prototype, we are to report findings and recommendations that will serve as a starting point on how, at least, parts of this kind of system could be constructed.

1.4 Limitations

Focus is on how the computations can be made in a distributed fashion rather than how the book keeping communication of the distributed system is handled.

The most important metric is assumed to be short response time due to the fact that visualization tools are heavily dependent on speed to be useful. The single user case is in focus, meaning that overall low latency for many users/jobs is not primarily considered.

The prototype will not handle data sizes around petabytes, as mentioned in the background. This is due to that we simply do not have the resources needed to setup a distributed system of the size needed, and therefore cannot test it.

The prototype will assume that all data will fit in the memory of its host, and most analysis around it will be based on this assumption. Deep investigation of a disk-based solution, probably taking advantage of striping, will be left for future work.

1.5 Description of remaining sections

In Section 2 we will analyze what is wanted from such a system, what existing algorithms can give (primarily in speed) and discuss the wanted aspects of the system as a distributed system. Section 3 will cover what we implemented, while Section 4 handles a performance test of our system in comparison to other alternatives. Section 5 discusses the results and tries to put them in a perspective.

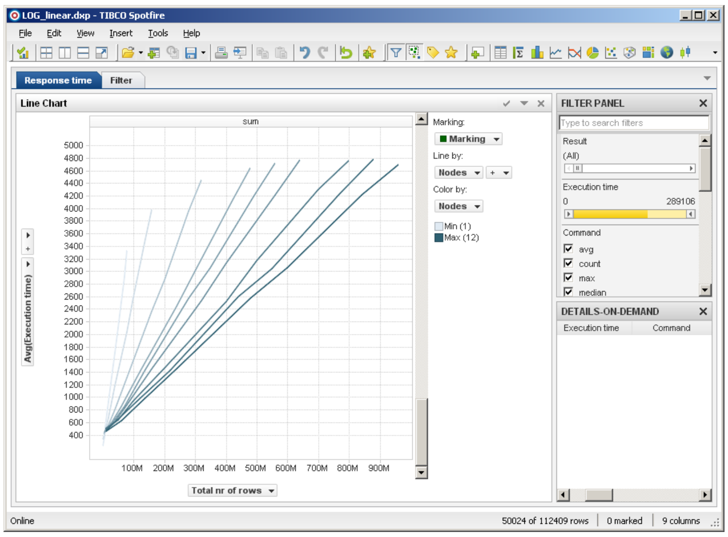

Figure 3: Screenshot from the visualization tool TIBCO Spotfire

2 Analysis

The domain for the system for distributed calculations is a visualization tool. A visualization tool reads data, such as a table, from a source, and then presents it visually to a user. Figure 3 shows a screenshot from a visualization tool, TIBCO Spotfire.

Today, the TIBCO Spotfire client handles large data in a straight forward fashion, ie. when the memory is no longer sufficient to hold the data, the data is paged out onto the disk. Since disk access is orders of magnitude slower than memory access, response times increase drastically once this barrier is hit.

This section will contain an analysis of what is needed from a system for distributed calculations that is to overcome this problem. What are the overall demands of the system, that is, how the system must behave towards a user? How can the system be implemented to fulfill these demands, and what problems and limitations will such an implementation lead to?

2.1 Constraints from the visualization domain

Since the visualization tool is based around the idea of having a human to operate it, it must be fast. A human analyst will loose his chain of thoughts if he has to wait for many minutes up to hours every time he want to try something out, thus yielding a poorer analysis. In order to meet the analyst's demands, Spotfire has chosen to implement their visualization tool completely in-memory.

When scaling up the size of the data sets being analyzed, the memory roof will soon be hit, forcing a spill onto disks. Main memory is also slow compared to CPU speed [20], and the gap between them is increasing, but disks are even

Manufacturer	Model	Year	Litre petrol per mile
Saab	95	1998	155
Volvo	XC90	2008	300

Table 1: An example table

orders of magnitude slower than main memory, which makes a switch of data source from memory to disks rather unattractive.

By using the approach of a set of computers to do the calculations, the combined size of memory will be the sum of all individual memory sizes, and the combined read speed from disk will also approach the sum of all individual disk read speeds. The resulting wall clock times will hopefully be much less than using a single computer disk based approach, assuming that the computations are not inherently sequential due to for example data dependencies. In Section 2.2.2, the partitioning of data and its effects on the communication overhead is analyzed.

Another limiting factor when considering a system for distributed calculations is data communication, although communication will be shown (Section 2.5.1) to be limited for the type of aggregations, that are a common type of queries in the system. Adding more processing units will naturally increase the amount of computations per amount of time that the system as a whole can perform.

2.2 What the data looks like

How the data to be analyzed is structured together with the assumed usage patterns will make a good start for analyzing models for storing and retrieving data.

First of all, it should be noted that in data analysis like this, there is none or very little modification of the data. This fact drastically limits what the system is going to have to support. Without updates, no transactions or re-building of possible indexes will be needed, making a complete database system overkill.

As for the structure of the data, a table based model is assumed. A table can be seen as a matrix M (shown in Figure 4 of dimension m x n, where m is the number of rows and n is the number of columns. Columns are typed and assumed to require a fixed number of bits for representation. An example of a table containing information about cars can be found in Table 1.

Obviously, the data has to be split up and spread out among the computers in the system in order to be processed in parallel. This will be described in detail in Section 2.2.2.

Interaction with the table takes place through queries, which will be described in the following section.

2.2.1 Queries

Tables can either be queried themselves or linked together, which is often called joined in the relational model. The effect of joins will be discussed in Section

$$\begin{bmatrix} a_{0,0} & \cdots & a_{0,m} \\ \vdots & \ddots & \vdots \\ a_{n,0} & \cdots & a_{n,m} \end{bmatrix}$$

Figure 4: An example of a table described as a matrix

2.2.3, and the term queries will be used for single table queries henceforth if not specifically stated otherwise.

A query on a table might include combinations of aggregations and constraints, specifying what is to be returned. There are of course many possible languages that can be used for querying, but there is a commonly used standard to use SQL (or SEQUEL in its old form) [10] [18]. An example query in the language SQL with the result being the average year of production of the cars in the table, found in Figure 1, might look like the following.

SELECT AVG(Year) FROM Cars;

yielding the result 03. Queries can be much more complicated, using groupings and orderings on the result.

2.2.2 How to partition the data

Since the data is to be processed by a set of computers which do not share any memory, the data has to be partitioned among the nodes. The idea is simply to partition the data into a disjoint set of parts, hereby called chunks, that will be distributed among the nodes in the system. The chunks could be replicated and put on many of the nodes as a way to ensure that no data will be lost, with a reasonably high probability, due to for example a node going down. This section, however, will deal with different ways to split the data and what the consequences in form of communication and computation time might be.

The term storage means different things depending on how the nodes are configured. If a node is configured to hold all data in the memory only, then storage refers to the memory. If a node is configured to read the data from disk, eg. from a striped array of disks, then storage refers to these disks.

Two immediate possibilities for partitioning the table formed data is to split it column wise or row wise. The splitting strategy will not affect the number of nodes needed with regard to total storage capacity, but it may affect the number of messages sent to coordinate the calculation as well as the local computation time on a node.

Consider the data held in a table described by the matrix M of size m x n, where m is the number of rows and n the number of columns. For every column's index and for every pair of rows, the type, and hence the size needed to store the value, is the same by the definition of a table above. The maximum amount of data that a node can hold in its storage is called R. If the choice is made to split the data row-wise, every one of the P nodes will hold a maximum of $\frac{mnc}{R}$ rows of data, where c is the size needed to store a value (assumed the same for every column). If, on the other hand, the choice is made to split column-wise, every node will hold a maximum of $\frac{mc}{R}$ rows of data.

When computing something based solemnly on one column, under the assumptions that the position of the data is known in advance, direct connections between nodes is always possible and that the result of the query is small enough to fit in one message, the first splitting strategy will need to send at least $\frac{2mc}{R}$ messages per query; one per node initiating the calculation and one per node returning the result. The column wise splitting will only need $\frac{2mc}{R}$ messages, since only $\frac{mc}{R}$ nodes hold the data of interest, which is a factor n less than the first strategy.

Performing a query where several columns are concerned will also need a different amount of communication. Using the row wise splitting, every query involving only the columns of that row and constants can be performed on a row at a time in parallel over the system, no more messages than said in the above paragraph will be needed. However, when splitting column wise, a lot more information needs to be transferred between nodes. When all the nodes containing a certain column have found which "rows" that passes the predicate concerning that column, the information has to be transferred to the nodes handling the column or columns to be returned by the query. This communication is quite costly as the number of messages/information sent between the nodes in worst case are O(m). The communication also increases linearly by the number of columns used in predicates.

The amount of computational work that each node needs to perform is also affected by the splitting. In the above reasoning, the column wise splitting will result in the nodes having to consider n times as many values as in the row wise splitting. Clearly, the ratio between local computation time and the time it takes to take care of communication must be consider in order to achieve the lowest overall response time.

To know in advance where all data is positioned does not came for free. Communication must take place in order to ensure this information is known to all nodes at all times, otherwise the assumption of direct communication cannot be used. To handle this kind of information, and the cost of it will not be analyzed here

The choice of how to split the data has been a hot area in database research lately [3]. To split column wise, using what is called a *column-store*, has been claimed to be a lot faster [30], especially in databases optimized for reading. In [3], a comparison between row-stores and column-stores is performed, and the reason for speedup of using column-stores are investigated. Certain optimizations used in column-stores are ascribed a lot of the speedup; compression of data and what is called *Late materialization* in particular. Late materialization is about merging columns to the desired tuple, where every row is a tuple of columns, from the different columns as late as possible. By using bitfields or similar means to hold information about what rows to include, one can traverse any columns used in a predicate/filter and only at the last moment construct the tuples that are to be sent back to the user. This way, according to the paper, tuples which are not to be used will not be unnecessarily created.

[20] has investigated how the caches are affected by the different splitting strategies. It starts by concluding that memory access is lagging behind in speed in comparison to the CPU. Caches, the technology used to hide the effect of the slow memory, is utilized to different degrees in database systems depending on the splitting strategy. The paper concludes that database operations, especially linear scans of the data, benefits much more from caches if the data is split in a column wise fashion.

The column-store approach discussed above has been implemented in several systems [30] [9], but none of these systems, or any that we are aware of, have dealt with the case of a distributed column-store. In the case of a local database system, the network communication is non-existing, whereas in the distributed case, column-stores would require quite some communication in comparison to row-stores according to the reasoning above.

We expect that the communication overhead of having a distributed column-store will outweigh any benefits gained (in for example cache use and others) in comparison to a row-store. Since data sending over a network is likely to require some memory access, it is definitely a more costly operation than simply fetching memory into caches. Note, though, that this has not been properly tested by us or by any source that we have found.

One could imagine, for the distributed case, to partition the data row-wise between the nodes and have them used in a column-wise way internally in every node. By doing this, every single node may use the optimizations developed for column-stores, while being used in a distributed way. Advanced database operations, like tables linked together, according to the relational model, using joins will require a lot communication anyhow, but they will be dealt with separately in Section 2.2.3, but for other operations this combination looks promising.

2.2.3 Joins

Introduction
Join is a binary relation from the relational algebra, which is a part of the field of first-order logic. Join takes two relations (tables in this context) and returns a set of all combinations of tuples (rows in this context), where attributes (posts in this context) are equal.

In SQL, there is an operation JOIN that takes two tables and a predicate, and returns a fused table based on the predicate. There are four kinds of JOINs in standard SQL: INNER, OUTER, LEFT and RIGHT JOINs. While different in their details, all kind of joins pose the same problem to a database-like environment, and it is even more punishing in a distributed environment. The most common type of JOIN is the INNER JOIN. We illustrate it with an example with two tables where the first table contains merchandise, represented by a name of the product and an id of the store where it can be bought. The second table contains the stores, identified by the store id and the address of the store. For simplification, note that every piece of merchandise is sold at one store only.

Table 2: Table merchandise

Name	StoreID
Hat	1
Shoe	2
Umbrella	1
Gloves	3

Table 3: Table stores

storeID	Address
1	Cremona, Chalmers, Gothenburg
2	Spotfire, First Longstreet, Gothenburg
3	TIBCO HQ, Palo Alto, USA

An SQL query using the standard INNER JOIN will return at what address the individual pieces of merchandise can be bought. The query can be formulated in two ways, both of which produce the exact same result.

```
SELECT merchandise.Name,stores.Address
FROM merchandise INNER JOIN stores ON
merchandise.storeID = stores.storeID
```

is equivalent to the, possibly more intuitive,

```
SELECT merchandise.Name, stores.Address
FROM merchandise, stores
WHERE merchandise.storeID = stores.storeID
```

The ensuing result will look as follows

Table 4: Table resulting from a join operation

merchandise.Name	stores.Address
Hat	Cremona, Chalmers, Gothenburg
Sheo	Spotfire, First Loongstreet, Gothenburg
Umbrella	Cremona, Chalmers, Gothenburg
Gloves	TIBCO HQ, Palo Alto, USA

Problem

The issue with dealing with joins in a distributed system is that it no longer suffices to compare with local chunks only. Filters that compare only to literal values need only compare to values on the same row, eg.

```
SELECT Name FROM merchandise, stores WHERE storeID = 2
```

In a join, all values in one column need to be compared to all values in the other column. If the columns are spread out in a network, this means a lot of traffic to build the new table, and much traffic means long response times.

Solutions

The solution to this problem is divided into two main categories.

1. When is the new table constructed
2. How is this new table constructed.

The first question is the smallest, as it has in essence only two possible answers. The new table can be built on-demand, i.e. when the query is being processed. The other alternative is that all desired "join tables" are materialized in a preprocessing step, and are given names. Further queries refer to these "join tables" and queries that include joins are prohibited.

Both of these approaches have their own advantages and disadvantages. The on-demand solution is more flexible, and does not require for the user to know in advance that he or she might want to study a relation that might not be obvious at first. The preprocessing solution has the advantage of being able to actually produce reasonable response times on large data set. A special case of joining is to join aggregated tables, which could be performed locally by the client upon receiving the tables from the system. The computation times for this join is likely to be pretty short, since the aggregated table is likely to be of manageable size. A query on a large amount of data that does include a join will take a lot more time, probably too long to be useful in this context of a visualization tool. If the preprocessing solution is chosen, the user will have to predefine an interesting relation and then wait for the new table to be constructed. Further queries on this relation, such as changing filters and switching aggregation, will take the same time as queries on an ordinary table. The conclusion will have to be that only prematerialized joins are reasonable in this context.

There are several algorithms to perform joins, some of which are described in [3]. Describing and analyzing these algorithms is not the focus of this report, since our conclusion is that joins will have to be prematerialized. This means that a join will not affect the response time of a query.

2.3 Consequences of big data sets

The size of the data set being analyzed affects how the analyst will work with the data. Big data sets will put the analyst in the position of having to use only aggregations or very limited sample selections for finding for example correlations in the data.

In the case of a small data set that can easily fit on a single computer, the analyst might do a visualization of the data for example in the form of a table sorted on a column that the analyst finds especially interesting, and see what the top ten rows contain. Performing the sorting calculation required by this query will likely not take many seconds. Had the data set been really big and spread out on a number of nodes in a network, then the question would likely have taken considerably longer.

The key to avoiding such unnecessary long waiting time lie in limiting the number of rows affected, or using only aggregations. Had the analyst specified that he would only be interested in the top segment of the sorted result, then the top segment could have simply been fetched and sorted locally before presenting it

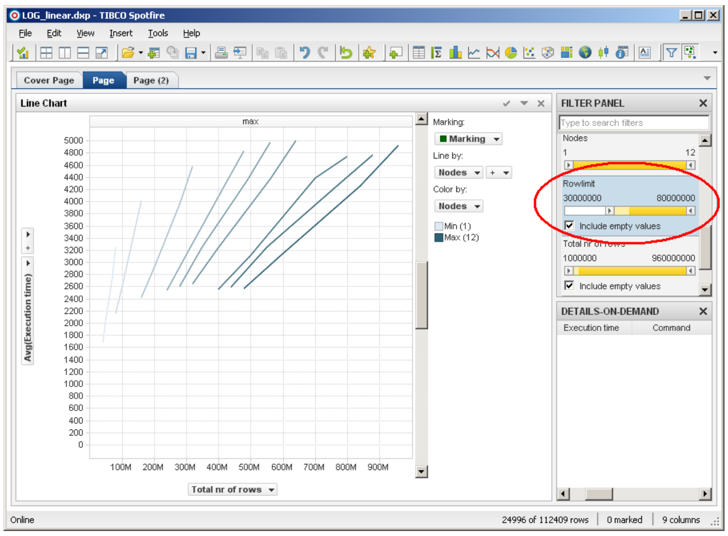

Figure 5: A filter is being applied in the TIBCO Spotfire client

to the user, requiring a lot less network traffic than the sorting, and thereby likely providing the results faster.

Plotting a two dimensional diagram where all elements in the data set are represented by marker on the diagram will work fine in the case of small data sets, but as the data set grows larger, and the diagram needs to use all data points in order to draw the diagram, the response time will simply be to big. By instead using something like a density plot, with a limited resolution, the analyst will get an overview where big trends will likely show up. Smaller differences are unfortunately likely to be masked when using this approximating function.

2.4 Problems of interest

Certain statistical measures are more common than others when visualizing the data. In this section, the most common categories of problems that one wants to solve will be discussed.

2.4.1 Filtering

Since the purpose for performing the calculations that this report covers is to be able to visualize them, an important feature is the filter. A filter cuts out rows from being included in a query, based on some predicate.

A filter may compare values to either literals, as in 5, or to values of other columns on the same row. Filters allows the analyst to eg. cut out extreme values from the data set, that would otherwise distort aggregations such as the average. It allows for queries to deal with specific intervals or categories, which may be of interest.

2.4.2 Aggregations

An aggregation is a function that given a collection of inputs, e.g. in the form of a list of values, returns a single value result that is based on this collection. This result is of a known size [23] regardless of the size of the input, why e.g. sorting and finding all unique values does not count as an aggregation.

The result of a simple aggregation on data can be both an important result in itself and a value needed only by the visualization tool in order to give the user an overview of options. For example, the maximum and minimum value of a certain column is often wanted in order to present a user with a relevant filtering interface.

In [23], aggregations are classified into five groups; distributive, algebraic, holistic, unique and content-sensitive. The first group, the distributive aggregations, consists of max, min, sum and count. Algebraic aggregations are plus, minus, average and variance. Median, which is described below, is a member of the holistic category and unique is in the unique category. All of the computations described above is of interest to the visualization tool.

Median
The median of a collection is the middle ranked element of that collection. Its value is intuitively similar to the average, but they are not the same. As an example, take the collection [1, 1, 1, 20, 27], where order does not matter, but presented in ascending order for sake of simplicity. The average of this collection is (1 + 1 + 1 + 20 + 27)/5 = 50/5 = 10. The median, as said, is the middle ranked element, thus in this case it is the third element which is 1. The median is sometimes called the second quartile. The first quartile is the element with rank n/4 and the third quartile is the element with rank 3n/4.

Obtaining the median or the other quartiles, or indeed any number at a given rank in an unsorted list is a significantly harder task compared to for example finding the average [21]. Given that they are less easily affected by outstanding extremes compared to the average, they are in many cases also more interesting. As the data sample shrinks, the effects of extreme values, caused by eg. a transient fault such as a startup procedure, may affect the average gravely without it having a real meaning of the long-term expected value.

The quartiles, the first one being the n/4:th smallest element of n elements and the third one being the 3n/4:th smallest element of n elements, give a hint of how spread out the domain of the values are.

K-selection
The problem of finding the number of a certain rank k, the kth smallest value, in an unordered set of size n is called k-selection. It has been known to be solvable in linear time in terms of the input size since at least the 70s [8]. Finding the median value in a set would correspond to finding the value of rank $\frac{n}{2}$, and similarly is true for quartiles, which means that k-selection is more general than the problem of finding the median.

Unique
Formally, unique is not an aggregation, since the size of its output, a set of items, may depend on the size of the input. Given a list of random floating point

numbers in a specific range, each element is possibly unique, hence the resulting set may be just a permutation of the input. The aggregation class called unique contains aggregations such as COUNT DISTINCT in SQL, which returns the number of unique elements, rather then the elements themselves. Despite not being a "pure" aggregation according to the definition above (aggregations), in database documentation, unique is often listed together with aggregations such as sum, avg and count, hence we include it in this section.

To exemplify the unique function, we take a collection of elements [1, 2, 2, 3, 3, 5]. There are in total four unique values in this collection, and these are [1, 2, 3, 5]. Note that the order of the elements in both the input collection and the answer is irrelevant, and they are presented in a sorted order for the sake of simplicity.

Identifying the set of unique values in a given data set can be helpful if the number of unique values are manageable, for a human or a computer. Eg, the set of unique numbers in a tall column of integers with values ranging from 0 to 100 will probably contain a lot fewer unique values than a tall column of floats with values ranging from 0 to 1.

The set of unique values are used for example in queries with grouping. For each unique value in one column, the query is applied to another (or the same) column. Other uses of the unique value query could be

Standard deviation
The standard deviation, denoted σ, gives an easily interpreted measurement of how spread out the value domain is. It is defined as $\sigma = i\sqrt{E[(X-\mu)^2]}$ where $E[X]$ is the expected value of the random variable X and μ is the mean value. Thus, the standard deviation tells how much a value is expected to deviate from the mean value assuming a normal distribution.

As an example, with the set $[3, 3, 7, 7,]$, the average is 5. One can also easily see that the average deviation from the average in the set is 2, since all four elements have a distance, in absolute numbers, of 2 from 5. A bit more formally, $\sigma = \sqrt{(2^2 + 2^2 + 2^2 + 2^2)/4} = 2$.

When visualizing sampled data with a normal distribution of values, the standard deviation may help as an indicator of whether enough data has been collected. If the data sample is expected to be normally distributed, or relatively close to it, a high standard deviation may be an indication that not enough data has been collected, and aggregations such as the average will be highly affected by for example transient faults such as a startup procedure.

Having the same unit as the random variable, standard deviation is easily comprehended by a human, and it is therefore important to be able to visualize it with an appropriate visualization.

Another measure of variability is variance. It is defined as $\mathrm{Var(X)} = E[(X-\mu)^2]$, and is thus the standard deviation squared, which means that the unit of the variance is not the same as the random variable.

2.4.3 Indexing

In the world of databases, indexing plays the same role as with a book. Instead of having to look at each page of the book to find the desired chapter, the reader looks for a keyword in the index and goes directly to the page that contains that keyword.

In the context of visualization, an index may help operations such as filtering and certain aggregations. If the filter is set to only show values where the value of an indexed column is in a certain range, the system may directly collect these values by looking in the index.

There exists several types of indexes, but an in-depth analysis is not within the scope of this report. One way of implementing an index [2] is to create a map where each unique value of a column represents a key, and the value is a list of positions in the data where that value can be found. Using this implementation, apart from filterings, operations such as obtaining the minimum value or the maximum value can be sped up considerably by just looking in the index.

If the data is to be held in memory, the gain of indexes is not that big [16]. Disk based systems gain much from indexes since each I/O operation costs much. Indexes will not be analyzed further, but is left as possible further work.

2.4.4 Clustering

To be able to find natural groups in the data can be very helpful for an analyst. For example, when marketing a product it would be much more easy to handle a few groups of customers with similar features than to handle all customers individually.

Clustering is, basically put, the problem of grouping data into disjoint groups according to some notion of similarity, like in the example above. It is a problem that has been well studied during the last decades. An overview of clustering can be found in [19], where different models are classified and compared. Not too surprisingly, humans are pretty good at clustering in two ore three dimensions, but in more abstract spaces humans are no match for computers, according to [19].

Clustering algorithms are split into two main categories in [19]: hierarchical and partitional. Hierarchical clustering yields what is called a dendrogram, which is a tree of clusters and information about at which level of similarity two clusters are merged to one. Partitional clustering will result in only one clustering.

All clustering algorithms are based on comparison of similarity between data points, and there are several classes of such comparison functions, which we will not go into details about here.

2.4.5 Sorting

To sort data is one of the most common things done by computers. However, as discussed in Section 2.3, the size of the data set affects what will likely be useful in a visualization tool. If querying for a subset of the columns from all rows sorted by a certain column, the query will need to not only sort all the data but also to send it to the computer asking for it. For big datasets, even the final sending of the sorted data will likely take so much time that it is not really suited for a visualization tools.

More often the analyst just needs the top or bottom k values, which is a much simpler problem for $k \ll n$, where n is the total number of rows. As long as only small subsets of the data need to be returned, the problem becomes much easier. For really small k, one can even just request all rows of rank higher than k and perform the sorting locally on the receiving node.

The problem of sorting all the data is sometimes still a valid thing to ask for, even for a visualization software. That is, there are cases where this is exactly what the user wants to do. However, since the visualization system is built with response time in focus, sorting could possibly be done outside of the system.

2.5 Structures for communication

There exists several structures for communication that many distributed algorithms rely upon. These are a few of them that appear frequently in scientific articles, and around which many algorithms are constructed around.

2.5.1 Convergecast

Convergecast is not an algorithm for a specific problem, but a way of communicating used in many calculations. Convergecast is a form of flooding algorithm [22] that can be applied if the network is in the form of a tree. The algorithm provides a scheme for communicating the command and identification of the dataset to use to all nodes in the network, and also to collect the answer. The converge algorithm will be described in the context of performing aggregations below.

The convergecast algorithm starts by the root node sending out the command to its children. All nodes receiving the message forwards it onward to its children until it reaches a leaf (a node with only one neighbor). The leaf performs what it is told to do and sends the result back to its parent. A parent node that has relayed the message to all its children starts to perform its work on the local data set, awaiting the replies from all children. When all replies has come back, the node performs the aggregating function on all partial answers and then sends the result to its parent, which does the same until the answers reaches the root node.

Other variations of similar techniques, where the Propagation of Information with Feedback algorithm, PIF, accomplishes the same goal but without the explicit tree structure.

2.5.2 Message Passing Interface

Message Passing Interface, hereafter MPI, is a communications protocol used for programming parallel computers. MPI 1.0 was released in June 1994 [24] by the Message Passing Interface Forum, consisting of over 40 organizations with a shared goal of creating a standardized message passing interface specification independent from a programming language, although official bindings exist for C and Fortran. MPI is now a de facto standard for message passing, and it is often used in both scientific articles and in real life with high-performance computing. [7]

From a programmer's view, MPI is a set of functions that handles communication in a black box way. Given a correct implementation, the programmer does not have to think about how the communication works on a low level, but it is still necessary for the programmer to handle situations such as starvation. The communication primitives defined in MPI handle both pear-to-peer communication as well as one-to-many and many-to-one communication structures. Every communication call may be synchronous or asynchronous, and they may be blocking or non-blocking.

For peer-to-peer communication, there are the MPI_Send and MPI_Receive functions. These exist in blocking/non-blocking and synchronous/asynchronous versions. For one-to-many and many-to-one communication, the functions also exist in blocking/non-blocking and synchronous/asynchronous versions. These functions split or collects the data in a predefined pattern. MPI_Bcast sends a message from one process to all other processes. MPI_Scatter splits the message, such as a list on one process, so that each process gets a peace of the list. MPI_Gather collects elements on all processes and creates a collection of those elements on a specified processor. MPI_Allgather works as MPI_Gather, with the difference that all processes gets the resulting collection. MPI_Reduce works quite similar to MPI_Gather as well, but it also applies one of the predefined reducers on the ensuing list, such as calculating the sum of all elements in a list of integers. The sum is then what the programmer receives, and the list is never visible.

2.6 Algorithms of interest

All algorithms in the categories distributive and algebraic aggregations can be solved with convergecast in a distributed network [22]. In the same paper, k-selection in a network of diameter D, where the diameter is the maximum distance between any two nodes, is proved to have a lower bound of $\Omega(D \log_d n)$ messages sent, which makes it more complex than convergecast. The problem of finding the unique values also needs some special attention, which will be discussed in Section 2.6.2.

2.6.1 K-selection algorithms

Several algorithms, both randomized and deterministic have been developed since the publication of [8] and several versions have been translated to use many processing units in parallel [28]. Below follows a description of one of the parallel algorithms, K-select 1. Its main features is its simplicity combined with relatively good performance. There exists some specialized versions of the k-select algorithms, one is [6] which has a worst case complexity of $O(\frac{n}{p}) \log\log n + (\tau + \sigma) \log p \log\log n$, where n is the number of elements, p is the number of computational units and τ and σ are network speed constants.

The algorithm

K-selection 1 is a distributive version of the first of two algorithms from [4] called k-SELECT (hereafter, k-SELECT means this first of the two k-SELECT algorithms). k-SELECT was originally a sequential probabilistic algorithm with a linear average complexity, and a quadratic worst-case complexity [28].

Let us first describe the sequential algorithm. It is very similar to the distributed version. B, in this algorithm, is the set of all numbers, called a bag. The algorithm finds the k:th smallest element in B, so the to find the median, k is set to $|B|/2$. The algorithms consists of three steps.

Step 1 - Random element Randomly, choose an element m from B.

Step 2 - Partition Partition B into three subbags. BL contains all elements that are smaller than m, BE contains all elements that are equal to m, and BG contains all elements that are greater than m. Define B' to be BL, BE or BG according to these rules:
$B' = BL$ if $k = |BL|$
$B' = BE$ if $|BL| < k \leq |BL| + |BE|$
$B' = BG$ if $k > |BL| + |BE|$

If $B' = BE$, than return m as the result. Otherwise, compute k' according to these rules:
$k' = k$ if $B = BL$
$k' = k - |BL| - |BE|$ if $B' = BG$

Step 3 - Recursion Apply the algorithm recursively to find the k':th element in B'.

The distributed implementation of this algorithm follows all the same rules as the sequential version. Both the probabilistic and the worst-case complexities are the same as with the sequential version. All steps in the distributed version are direct translations of the sequential version, with added communication. We assume that there already exists a functioning network N = (P,C) where P is the set of processes and C is the set of channels. All processes in P have their own bag B and a set (which may be empty) of subtrees, which are processes that has P between themselves and root (see step 0 for the explanation of this).

Step 0 Create a rooted spanning tree from N. There are several papers discussing how to find such a structure distributedly, such as [15]. The root of the rooted spanning tree is hereafter called just root.

Step 1 - Random element Apart from knowing the size of its own, local bag, denoted t, all nodes know the size of all their subtrees' bags, called t(1), t(2), ..., t(d) for each of their d subtrees. Thus, the sum of root's t plus the size of all root's subtrees' bags is the total number of elements in N, called n. Now, the task is to select one element at random. Root randomly chooses as integer i, from 1 to n, to represent the position of the random element in all of N. Now root has to decide whether to locate element number i locally or at a child. If i = t, then root has element number i locally. It calls element number i m, and goes on to step 2. If i t, i is in its f:th subtree. It sends LOCATE(j) to the f:th child, where j is the smallest positive integer of the form i - t - t(1) - ... - t(f-1). When a node receives LOCATE(j) it acts similarly to root, but sends m to its parent, which in turn passes it to its parent, all the way up to root.

Step 2 - Partition Once m is known by each node, each node starts to partition B into three subbags BL, BE and BG according to the same rules as for the sequential version of the algorithm. Root finds out the global sizes of $|BL|$, $|BE|$ and $|BG|$ using the same mechanism as in step 1. Root then chooses B' and k' according to the same rules as the sequential version and broadcasts its decision to all nodes. If B' = BE, m is returned as the answer, otherwise continue to step 3.

Step 3 - Recursion Apply the algorithm recursively on all nodes with k' and B' as parameters, starting at step 1.

Complexity
Since this is a distributed algorithm, complexity may be calculated in two ways. Often when analyzing distributed systems, the time required for local computations are ignored, and all cost is attributed to communication. Thus, complexity, in a distributed context, often means the number of messages that has to be sent for the algorithm to produce its answer. In the following analysis, D is the diameter of the network, n is the total number of elements in the network and p is the number of processors.

As shown in [28], the expected number of messages required to solve the median using k-select 1 is $O(|P| \log n)$.

[28] also claims that the number of rounds expected to complete a calculation is in the order of O(log(n)). On each round, there are essentially two operations that take any considerable time. These operations are counting the elements in step 1, and performing the partition in step 2. All other operations can he considered to take constant time. Counting the elements takes O(p*(n/p)) = O(n) time units. The partitioning goes through every element, just as the counting, but also adds a comparison. This comparison takes constant time, so also the partitioning takes O(n) time units.

2.6.2 Unique

The unique operation can be implemented using a set data structure. A set is a collection of distinct elements, which means that an element can exist in the set only once. If the number 4 is added to a set that originally contains all even numbers between 0 and 10, the set will remain unchanged, but if the number 11 is added, then the set will grow to include also number 11.

There exist several ways to implement a set, such as lists, arrays, a hash table or different kind of trees. We chose an implementation based on a hash table, based on its constant-time complexity for insertions [31]. A hash table works on (key, value) pairs. For every value, there exists a corresponding key. Using a hash table, looking up a value, supplying a key, can be performed in constant time regardless of the number of elements in the hash table, given that the hash table is well dimensioned. The key is hashed using a hashing function, and since the size of the hash is known, a known number of comparisons must be made to look up the value corresponding to that hash.

Implementing a set using a hash table is done by using the input value as a key in the hash table. The value inserted to the hash table is unimportant and can be nullary. If a lookup using the input as the key returns the nullary value, then the input already exists as a key in the hash table. If it returns nothing, then the input is inserted into the hash table. To return the set, all keys are returned.

To find all unique values in a distributed fashion, all nodes create their own hash set with their respective unique values. All nodes then send their set to a root node, that performs a union operation between all sets. The union operation between two hash sets is quite brute force. If set A is going to be unioned with set B, every element in B is added to A as an element. This means that with n unique values and m nodes, the complexity of performing a union with the set of unique values from every node is O(n*m).

2.6.3 Clustering algorithms

A common clustering is k-means clustering, which is a partitional algorithm for clustering data consisting of d-dimensions (d-tuples) into k clusters. The algorithm works by selecting k centroid points in the d-dimensional space according to some schema. Then, it assigns every data point to its closest centroid, where the distance function is the squared Euclidean distance between the points. New centroid points are then calculated as the point in the middle of the shape consisting of the points from the data bound to a certain old centroid. The process is repeated until it has reached some stop criterion or has stabilized itself. The algorithm uses local optimization, which means that there is no g. The choice of starting centroids will greatly affect the end result of the algorithm, and several heuristics have been proposed for choosing them. One should note that finding the global optimum is in general NP-hard [5], and the above description of an algorithm is based on local optimization and is thus not guaranteed to give the most optimal answer.

A distributed version of the k-means clustering algorithm could be found in [13]. The algorithm is written for the MPI model, described in Section 2.5.2, and their paper claims almost a linear speedup in terms of nodes compared to a single node calculation.

2.7 Interesting metrics to measure

Given that the results are calculated on a separate system, invisible to the user, *response time* is the most critical metric. As long as a user gets a correct answer in time, it is not as important with metrics such as *memory usage* or *network traffic*.

However, in a multiuser system, metrics such as memory usage and network traffic might severely affect the response time, since low memory usage and little network traffic might allow for the system to work on two different tasks at the same time. Throughout the report, response time is given the highest priority since the empirical studies are done on a single user system.

Another metric that might be interesting is *accuracy*. With a large data set, some inaccuracy can be expected due to eg. variable precision in float numbers.

2.8 Investigating Map/Reduce as a back-end system

We started out trying to investigate the needs behind the search for a distributed back-end for the visualization tool. Initially, it appeared that really, really big data sets were the main focus. Due to the hype lately about the "cloud computing" and some big companies, notably Google and Yahoo, use of a newly invented abstraction for distributed computation we started investigating it. The abstraction, called Map/Reduce [12], had an implementation available as open source in the form of Hadoop, which we spent some time getting to know.

2.8.1 Description of how Map/Reduce works

Introduced by Google in 1994, Map/Reduce is a combination of the functions map and reduce from the world of functional programming. Solving a problem using Map/Reduce is done in two stages: a map stage and a reduce stage. Each of these stages are highly parallelizable, thus this abstraction fits very well for a distributed system. The types of map and reduce help explain how they work.

Table 5: Types of the Map/Reduce user defined functions

map	(k1,k2)	→	[(k2,v2)]
reduce	(k2,[k2])	→	[v3]

Map takes a key-value pair (k1, v1) and returns a list of new key-value pairs [(k2, v2)]. Each instance of map, called a mapper, get its own key-value pair in and does not have any side-effects, so several mappers can run simultaneously on many computers. The returned list of every mapper is sent to the reduce stage.

In the reduce stage each instance of the reduce function, called a reducer, gets a key paired with a list of all the values that were coupled with the key in the map stage. With this key and all values, a third list of values is produced. Type type of reduce is thus (k2, [v2]) $\rightarrow$ [v3], and also this stage is highly parallelizable. The result of the whole Map/Reduce operation is all [v3] combined.

As an example, imagine the problem to count the number of occurrences of every word in a document. The types will have the same names as in the table above. Each mapper will receive a chunk of the document, where the key, k1, might be some identifier of what chunk it has received. K1 is irrelevant for this problem. The important variable is v1, which contains the actual text in that chunk. Now the mapper splits up the text into words, and for each word outputs the word as the key, and the integer 1 as value. This mapper will thus have the types (String, String) → [(String,Integer)] in Haskell syntax.

Each reducer will receive a word and a list of the integer 1. The task is simply to add all integers, with the value 1, into a sum which represents the total number of occurrences of the word. Each reducer then outputs its key and sum as a tuple in a singleton list.

In pseudo code, the algorithm could look like (code)

```
map(String chunkid, String document):
// key: chunk id
//value: document contents

for each word w in document:
    Output(w, 1);


reduce(String word, Iterator values):
// key: a word
// values: a list of counts

int result = 0;
for each v in values:
    result += v;

Output([(word,result)]);
```

2.8.2 Hadoop

Apart from Google's own, closed and unavailable implementation of Map/Reduce, the biggest actor on the Map/Reduce stage is Hadoop. It is maintained by the Apache Foundation, and its biggest contributor is Yahoo, which uses Hadoop as part of their web search services. Hadoop has won several awards and competitions, including the prestigious TeraByte Sort competition where, in 2008, 1 TB of data was sorted in 3:29 minutes [25].

The Hadoop framework is built in Java. Its features include a web based maintenance interface, a complete distributed file system, called HDFS for Hadoop Distributed File System, and work load balancing. However, it is tuned for a much larger scale than what is interesting in our problem. Map/Reduce works best when the size of its input is in the order of terabytes and when there are hundreds or thousands of CPUs in the network [11]. In practice, this means that simple aggregations such as sum all have response times in the range of tens of seconds, regardless of the size of the input. With growing size of the input data, the response time grows too fast to be usable for visualization that is supposed to be in near real-time.

A comparison with distributed databases, which uses a data model like the one described in Section 2.2, in paper [26] concludes that Hadoop is many times slower than the state of the art distributed databases that it was compared against.

2.8.3 Fitting the interesting problems onto the model

The generality of the programming model, with the possibility of using any kind of data as input and allowing almost any kind of manipulation of it, makes it very powerful. It is easy to see why Google chose to use this very general model as their needs include working with files of different formats and where results of jobs could be anything from a reverse indexes to be used in their search engine to image manipulation in Google Maps.

It could be worth noting that even though much of the information handled by the big search engines is crunched by a Map/Reduce system, the search engine (where response time is of great importance) is very likely accessing the data through some other means than using the Map/Reduce system.

3 Method

3.1 Work-flow

In parallel with getting to know the Hadoop system, described in 2.8.2, we started thinking about an upper limit on how long time that could be spent doing computations and creating the visualizations before the whole idea of having a human analyst take all decisions would fail. If say a query would take 10 minutes, then the analyst would have to wait for that time before she could continue her work. Clearly, either the computation times need to stay relatively low or other setups should probably be considered.

The team behind Hive clearly states in [14] that even relatively small and simple queries might take around 5-10 minutes, due to the nature of Hadoop. Since we had concluded that this kind of response times were not realistic in the setting of a visualization tool, and that it would be possible to implement distributed algorithms with good time complexity without the restrictions that the Map/Reduce abstraction imposed, the Hadoop approach was scratched.

Instead, we focused on creating a new model of how the data would be queried and how queries could be implemented efficiently. Using a model of materialized tables and limited queries, as described in Section 2.3, we decided to look for distributed algorithms that could be applied to this distributed setup.

Using the model described above, we started to create a prototype of the system so that it could be tested and integrated to the Spotfire client. The choice of programming language fell on Java [32] as the target computers that was to run the program had quite a variety of architectures and operating systems. By using a virtual machine language with widespread implementations we hoped to not reduce the already small number of possible computers available for use further. The implementation, and its motivations, is described below in Section 3.2, and the following performance tests are found in the results Section 4. The test setup is described in the result section as well.

Later, we looked at models allowing extended queries including joins, and how they would effect the system if allowed. Joins, regardless of which of the algorithms (that exists to our knowledge) one uses for implementing them, may cause lots of communication. Based on some simple calculations we came to the conclusion that joins are too time consuming to be performed in a single query. Instead, we decided to push the responsibility of joining to the data source, part 1 in Figure 2, and force all joins to be materialized before entering the processing system (part 2 in Figure 2). This might seem like a big downside, but it is necessary in order to keep response times in the order of a few seconds, making it practically useful in a visualization tool such as TIBCO Spotfire.

3.2 System summary

The prototype system was built almost exclusively to see what kind of response times that could be reached for different number of nodes, data sizes and algorithms. The reason for this was based on that analysis through visualization simple is not useful if the response times are too long. To see whether it was indeed feasible to use a distributed system at all, and try to spot trends of for example how more nodes vs more data per node affects response time, was more important than to have the system be fully distributed and handling all problems that a reliable distributed system may face.

Data was stored completely in memory in the prototype. This was done in order to minimize response times for the user, and disks are known to be lagging behind both the CPU and memory. (Memory is itself lagging behind the CPU, but disks are even worse).

The prototype system is a straight-forward server/client solution. Nodes do not know about each other except when the server tells them where and how to reach other nodes, and it only does so when an algorithm really needs it. For example, the algorithm implemented for finding the median value expects the nodes to be organized in the form of a tree, which the server takes care of by ”connecting” them to each other in a suitable order.

Several simplifying assumptions were made in order to keep the prototype size manageable.

- Nodes are assumed to be able to connect directly to every other node
- The server initiates every job, or every job originates from the server
- Only values of integer and floating point numbers are used

Also, some simplifications were made after inspecting the Spotfire client.

- Only a fixed number of bits are needed to represent a single value
- The data is assumed to not overflow for aggregations on nodes. The server, however, is to be able to merge the aggregations into a data structure that does not overflow

The following list of aggregation algorithms are implemented.

- sum
- min/max
- avg
- count
- unique
- median

3.2.1 Data handling

As was discussed in Section 2.2.2, the data has to be partitioned and placed on an appropriate nodes in the system. The partitioning was chosen to be made on a row wise basis into chunks of a certain size. This size was chosen in order for it to be small enough so that the data could fit in the primary memory of a node while not being unnecessary small. The choice was based on the need for extra communication when querying over several columns using column wise splitting. Locally, the data was splitted by row.

To avoid effects from sorted data, or patterns otherwise, that could affect the resulting response times, we have made sure that all rows are randomly position over the nodes.

3.3 The server

The server was implemented as a separate program and run on a machine with an address known to the clients. This simplification was done to ease the implementation but it should not imply much loss of generality, since the nodes in the distributed system could have elected a node to take this role using an election algorithm [33]. Of course, this solution will not have to handle for example splits in the system, since only one node is ever the leader.

A TCP connection was kept open at all times between the server and every node in the system since creating such a connection is a costly thing. Through this connection, commands and answers where sent. As mentioned in Section 3.1 the median finding algorithm required a tree form on the network, and connections within this tree is created as the first step of median finding.

3.3.1 Job management interface

To perform a job, one would simply connect to the server through a socket and send text commands syntactically similar to SEQUEL [10], although simplified, and then the answers would be sent back as text. The prototype does not at all handle relations between tables. Only one user was allowed to start jobs at a time, but there is no reason why several users cannot use the system simultaneously.

3.3.2 Server command interface

Apart from the commands for querying data, several meta-commands exists as well to simplify performance testing.

The *nodes* command lists all connected nodes, together with information and settings of that node. These properties are the number of enabled threads and if the node is enabled or not. The information is the hostname and port of the control channel to the node, and from where the node got its settings. The settings may have the default values; they may come from a history list on the server (which is used if the node has been restarted, thus retaining its old settings), or they may come from the node itself, should the server have been restarted.

The *enable* and *disable* commands enables or disables connected nodes. A disabled node does not participate in performing a query. This setting was used to measure how well an algorithm scales with the number of nodes connected.

The *rowlimit* command sets how many rows that should be included in upcoming queries. This command was used to measure how well an algorithm scales with the number of rows included.

The *time* command returns the execution time of the previous job, given in milliseconds. It is upon the outcome of this command that we base our performance measurement.

The *meta* command returns information about the table that is being used. The current implementation allows for only a single table to be used in the system, but this limitation comes only from the implementation.

3.4 The client

The client is a node in the network that performs computations when a command from the server is received. In order to get low response times, the client keeps its active chunk in memory.

Several threads may process disjunct parts of the active chunk in parallel in order to further lower response times. The default behavior of the client is to use as many threads as there are cores/processors on the computer. This feature was added due to the steady rise of multicore computers, and the following need two write parallel programs in order to harvest their power. Since almost every internal operation on the data chunk only works on a particular row at a time, due to the nature of the query and chosen splitting strategy, queries are almost embarrassingly easy and efficient to parallelize on a node.

3.5 TIBCO Spotfire integration

To be able to test and demonstrate the actual usability of the distributed system for calculations, an extension was created for the TIBCO Spotfire client. This extension was created with a great deal of help from our on-site supervisor Jonas Svensson, an employee at the company with many years of experience of programming on and for the TIBCO Spotfire client.

The extension uses our system as a data source. It connects to the server to which it sends queries and from which it receives answers. Note that it does not handle placing the data on the system, since that is something that our prototype does not handle.

3.6 Algorithms implemented

The aggregations that are supported in our prototype was built around two architectures: one for the median calculation and one for all the rest. This is because the k-selection algorithm requires a lot of communication in the intermediate steps whereas for the other aggregations, the result of each node's calculations can be easily combined by a central node to the correct answer in just one step.

3.6.1 Easy parallelizable

All available jobs except from the median job can be trivially parallelized.

Sum A server collects the partial sum from all nodes and returns the sum of all subsums.

Min A server collects the minimum element from all nodes and returns the smallest element of all those elements.

Max A server collects the minimum element from all nodes and returns the largest element of all those elements.

Avg A server collects a tuple of the sum and the number of elements from all nodes. It then sums all subsums and all number of elements. It then returns the sum divided by all elements.

Count A server collects the number of elements from all nodes and returns the sum of all the number of elements.

Unique A server collects the sets including each node's unique elements and unions that with an own set (starting with the empty set). It then returns the set.

3.6.2 K-selection algorithm

Our implementation of the k-selection algorithm starts with building a binary tree, as the implementation of the rooted spanning tree mentioned in (2.3.k-select). On each computer, one thread of the MedianSolver class is started for each core. Each MedianSolver will operate as an independent node in the binary tree. On computers with multiple cores, each MedianSolver thread will take care of an equally sized chunk of the data.

A MedianSolver holds one Node object which represents its place in the tree. The Node object can send and receive messages from its parent and children, and this is the channel of communication used by the algorithm.

While the original algorithm is recursive, our solution is built with a loop and a state. When the MedianSolver is started for the first time, it goes into the INIT state. There, the tree is built and each node gets information about who its parent is, columns of interest and the types of the columns of interest. When the tree is built and all nodes know the types of all columns of interest, each MedianSolver enters the READY state.

The READY state is the state where a MedianSolver receives either the start command and a filter, or the stop command. This is the state that the MedianSolver enters after the median has been found, and it can be restarted with a new filter, which is used with queries that has a Group By clause. If the MedianSolver receives that stop command, it propagates the message to its children, kills its Node object, exits the loop and turns off. If the MedianSolver receives the start command, it enters the COUNT state.

Entering the COUNT state, we have now reached step 1 in the algorithm. Each node counts the number of elements in its bag and gets the size of all its children's bags, if any. It then sends the sum of its local size and its children's size to its parent. Now, if it is the first round, root sets k to be the middle rank of all elements. Root then enters the LOCATE state. All other nodes enter the IDLE state.

The IDLE state is a state where nodes, except root that may never enter the IDLE state, wait for a command to either locate an element in its bag or to start to partition.

In the LOCATE, root starts with picking a random number, i, between 1 and the total number of elements in the whole tree. This number represents the position of the upcoming partition element, m. If root has m locally (see rules in (2.3.k-select)), it sets m to B(i). If root does not have m locally, it sends the locate i command to the appropriate child, waiting in the IDLE state and waits for the child to return m. A node that is not root acts similarly, with the difference that it does not pick i randomly, but instead receives it from its parent. At the end of this state, root enters the PARTITION state and any other node goes back to the IDLE stage.

Now entering step 2, the PARTITION stage is the heaviest part of this algorithm. After root has found out the global sizes of BL, BE and BG it determines what to do next. If the median has been found, it is returned to the median server and the finished command is sent to all the other nodes. All nodes then return to the READY state, waiting for either the start command or the stop command from the median server. If the median was not found, which is usually the case, root decides what bag to choose and calculates a new k. Entering step 3, a choose command is sent to all nodes and all nodes return to the COUNT state.

3.6.3 Filtering

The objective for our implementation of filtering was to fill our needs to be able to

1. Implement a *Group by* functionality
2. Limit the number of rows included in a query

Our filter is a symbolic filter built around an abstract class Predicate, with subclasses

- AndPredicate
- OrPredicate
- SinglePredicate
- TruePredicate

The Predicate abstract class has a boolean function match, that returns true if the predicate holds. Each Predicate (with the exception of TruePredicate) holds an instance of the abstract class Comparison, with subclasses

- IntComparison

- FloatComparison

Each Comparison holds a literal of the correct type and an Operator, which is an enumeration of the following operators

- LT
- LET
- EQ
- GT
- GET
- NEQ

The comparison is fed numerous values to compare the literal to, based on the Operator.

The AndPredicate and OrPredicate classes hold, apart from the Comparison, a second Predicate. For AndPredicate's match to return true, both the Comparison and the second Predicate's match must return true. For OrPredicate's match to return true, either the Comparison or the second Predicate's match must return true.

SinglePredicate only holds a Comparison, and TruePredicate's match function always returns true without any checks.

4 Results and discussion

The prototype was built mainly for two reasons; getting some actual wall clock times giving a hint on how performance would be and to do an integration to the TIBCO Spotfire client.

4.1 Performance testing setup

Here follows a description of the hardware setup and how the tests were performed.

4.1.1 Hardware setup

The test was performed on the offices' work-stations, which gives a rather big variance in their hardware setup. The computers range from single processor 1.5 GHz Intel P4 machines with 2 GB of RAM to 3 GHz Intel quad cores machines with 4-8 GB of RAM.

Since the result from all nodes aggregations are needed to create the final result, the slowest computer will limit the response time of the whole system. Because of this, all computers joining the group ran a test suite of local aggregations and based on the time it took they were either accepted into the group or told to shut down. This initial test was designed only to catch extreme outliers, as a reaction to an incident where a really slow computer completely ruined a night of testing.

The maximum number of nodes ever connected to the system did not exceed 16, but since so many nodes were only connected a very short time (due to an unfortunately complicated problem of setting fire wall settings), the maximum number of nodes simultaneously connected when running the real tests were limited to 12.

The computers are connected through a 100 Mbit network with unknown throughput.

4.1.2 Choosing active nodes

All tests are based on choosing a subset of the nodes connected at random and a limitation on the number of rows used per node. This way, we could emulate the effect of having a constant amount of data in the system spread out over a varying number of computers.

The reason for randomly choosing which nodes to choose for each query is to simulate a real scenario as much as possible. If all tests done on one node is performed on the same node, the resulting response times would be unrealistically stable. In a real-life scenario, it is not probable that the same one node is chosen every time. This is especially true if the system would allow several queries being processed in parallel.

During our tests, the *enable X* command enables X random nodes. This command would not be usable in a production environment, but it is of great help for testing, as it allows for easy measuring of how aggregations scale on the number of nodes in the network.

4.1.3 What our data looks like

All data used by the prototype was randomly generated before starting the client. This was partly because we did not want to flood the network with data communication and partly ta avoid unpredictable synergies to affect the resulting measurements. The table looked like the one in Table 6, with 3 columns total out of which one was held a floating point value and the other two were integers. The integers were encoded as 4 byte signed integers and the float was encoded as a single precision binary IEEE 754 [1] floating point numbers.

Table 6: Test table structure

Property	Column 1	Column 2	Column 3
Type	Integer	Float	Integer
Range	[0,14]	[0,1]	[0,19]

A total of 80 million rows were generated on each node using the Java pseudo random number generator. The first integer column was restricted to number in the interval [0,14] and the second one was limited to numbers in the interval [0,19]. The floating point column was limited to the interval [0,1]. The limits on the integer columns was added in order to have few enough unique values to be able to use the group by construction described in 3.3.2 on the columns.

4.2 Testing a database

There are of course many alternatives to using a distributed system. A database is a natural competitor for solving the problem since it closely matches the needs from the visualization tool. Databases may also reside upon a distributed network, but we made no tests of such a system, for several reasons. Primarily, there is the issue of time; our worked slipped more and more towards databases as work progressed, and at the time where we first considered a pure distributed database, there was no time to properly set up such a system and conduct meaningful tests.

In our tests of a database, we assume that the data is stored on the disk.

MySQL on single node
One of the computers used in the prototype was used as a database computer. The computer was chosen among the ones used throughout the testing, and its hardware characteristics are in the top part of the set of computers.

The computer was running CentOS 5, and MySQL was installed through the packet manager and was run using the out-of-the-box configuration. The version was "mysql Ver 14.12 Distrib 5.0.45, for redhat-linux-gnu (i686) using readline 5.0".

The table for use was created with the following command. In addition to the 3 columns used in the client, an additional key was needed.

```
 CREATE TABLE data (id int not null auto\_increment, primary
key(id), c1 int, c2 float, c3 int);
```

The table was then duplicated and an index was created on column c1 in the duplicate table data_indexed with the following command.

```
 CREATE INDEX index ON data\_indexed (c1);
```

Some simple commands were run and an average execution time was gathered. The variance was very small, so the number of tests was not very big. The resulting times measured is found in Table 7, together with a shortened version of the SQL-statement used

Table 7: MySQL performance results

Command	Column is indexed?	Execution time
SUM(c1)	Indexed	18.59 sec
SUM(c1)		17.29 sec
SUM(c2)		10.01 sec
DISTINCT(c1)	Indexed	0.01 sec
DISTINCT(c1)		47.40 sec
MAX(c1)	Indexed	0.01 sec
MAX(c1)		10.02 sec

As can be seen, going to a disk based system will result in quite some time taken in comparison to a pure in-memory solution. The time it takes to calculate the unique values for an unindexed column could be disregarded, as it would seem ridiculous to use in a real environment.

4.3 Testing our system

These are the results of the off-hours test runs of the prototype system built for this report. Additional graphs can be found in (Appendix X).

4.4 Comparing to the databases

Since the variance is quite big in our system, due to the fact that the slowest computer decides the response time of the whole operation, information about the standard deviation is included as well. The constant amount of 80 million rows are used here, to be able to directly compare against the database tests.

As can be seen in Figure 8, calculations involving one node has a standard deviation pretty close to the average execution time, which means that it is varying quite a bit. As more nodes are added, the standard deviation drops considerably (from around the same size as the average time to around 10average).

Table 8: MySQL performance results

Command	Nodes used	Avg time	Stddev	# of tests
SUM(c1)	1	1.52 s	1.56 s	248
SUM(c2)	1	3.33 s	1.58 s	248
DISTINCT(c1)	1	2.59 s	1.15 s	248
MAX(c1)	1	1.62 s	0.59 s	248
SUM(c1)	2	1.07 s	0.23 s	250
SUM(c2)	2	2.04 s	0.45 s	248
DISTINCT(c1)	2	1.88 s	0.49 s	250
MAX(c1)	2	1.17 s	0.29 s	250
SUM(c1)	8	0.64 s	0.056 s	426
SUM(c2)	8	0.88 s	0.088 s	423
DISTINCT(c1)	8	0.89 s	0.080 s	426
MAX(c1)	8	0.67 s	0.118 s	426

4.5 Scalability

Comparing response times for small data, consisting only of 80 M rows is not really that interesting. The prototype was created with billions of rows in mind, and the performance scalability as the number of rows is increased is more interesting. The maximum total number of rows actually tested was limited only

Using the constructs for setting the number of active nodes together with the possibility of choosing how many rows to use per node, using the features described in Section 3.3.2, the system could easily be tested in different setups.

The "Total nr of rows" value comes trivially from multiplying the enabled number of nodes with the current rowlimit value. The rowlimit setting sets the number of rows that each node handles in a query. The rest of the rows are ignored.

As clearly visible, the max and the sum aggregations scale really well with the number of nodes added (as can be seen in Figure 6 and 7). This trend is visible for all aggregations except for the median aggregation.

A good way to see how the algorithms themselves scale with the size of the input is to compare the response time to the rowlimit value. Figure 9 shows the count aggregation, and its pattern is similar in all other aggregations (again, except for the median). Another interesting thing to measure is the computation time as a function of the number of computational units. Figure 11 shows how computation time scales for a constant amount of data when adding computational units.

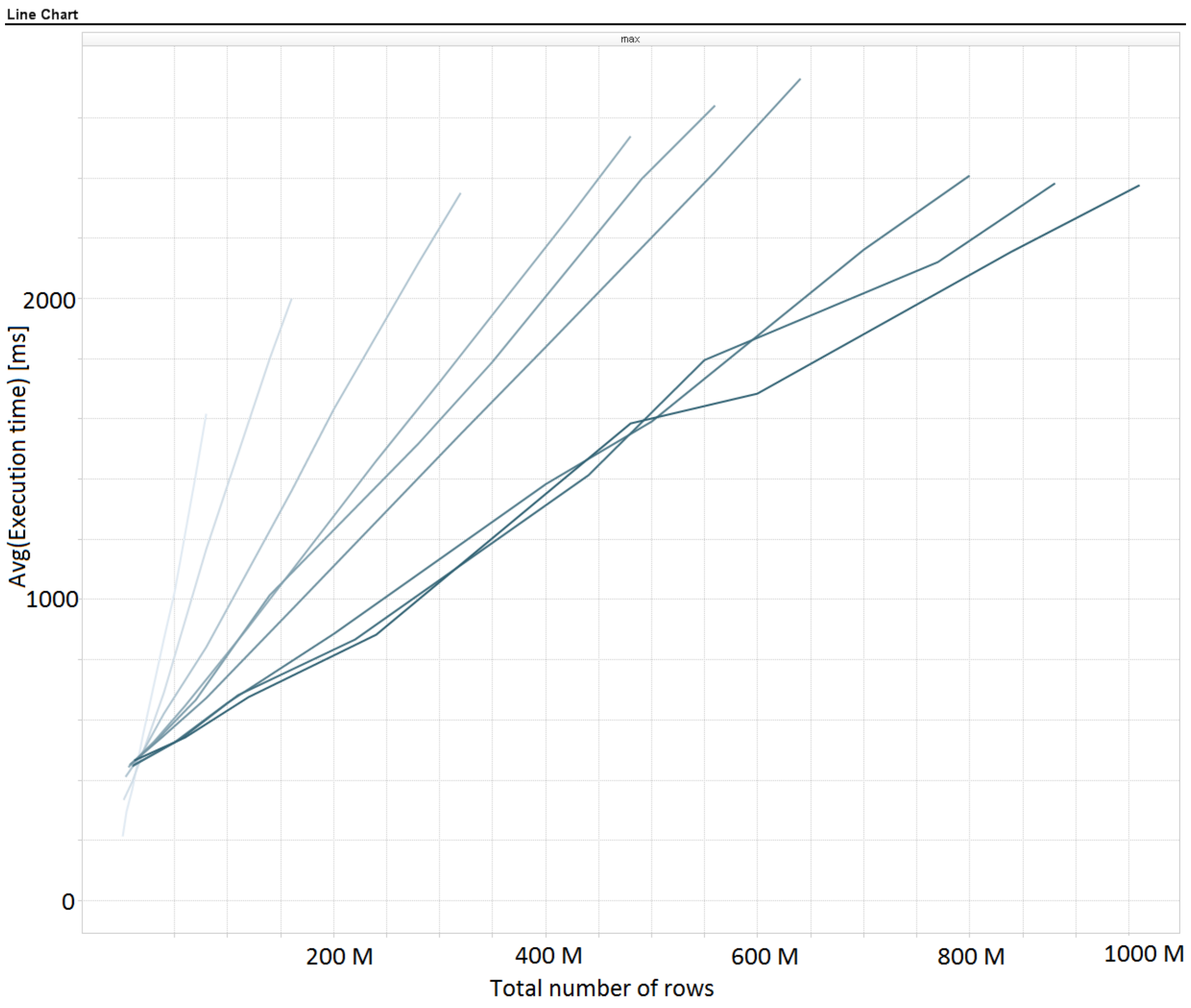

Figure 6: The max aggregation performed on Column 1 which consists of integers

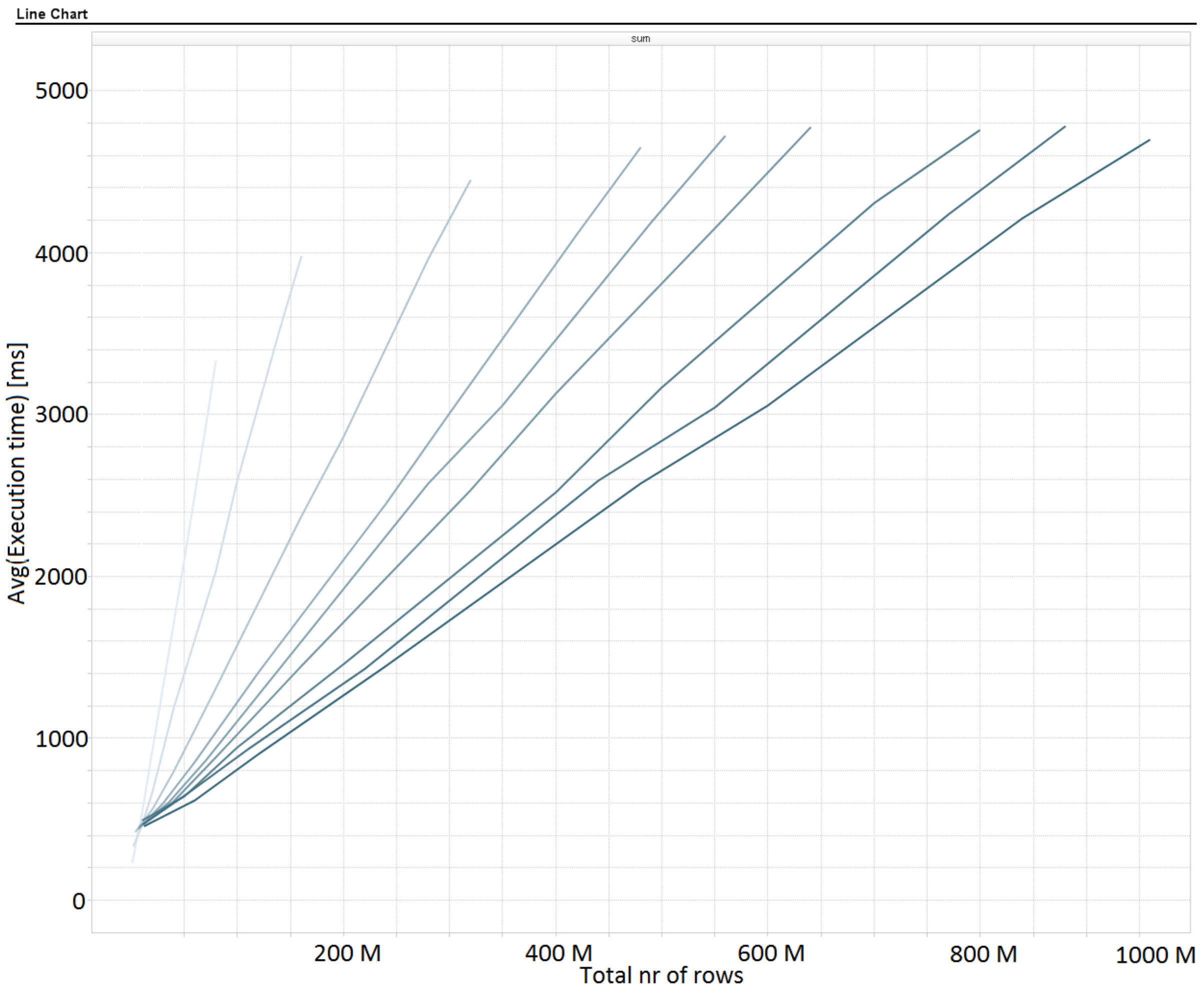

Figure 7: The sum aggregation performed on Column 2 which consists of floating point values

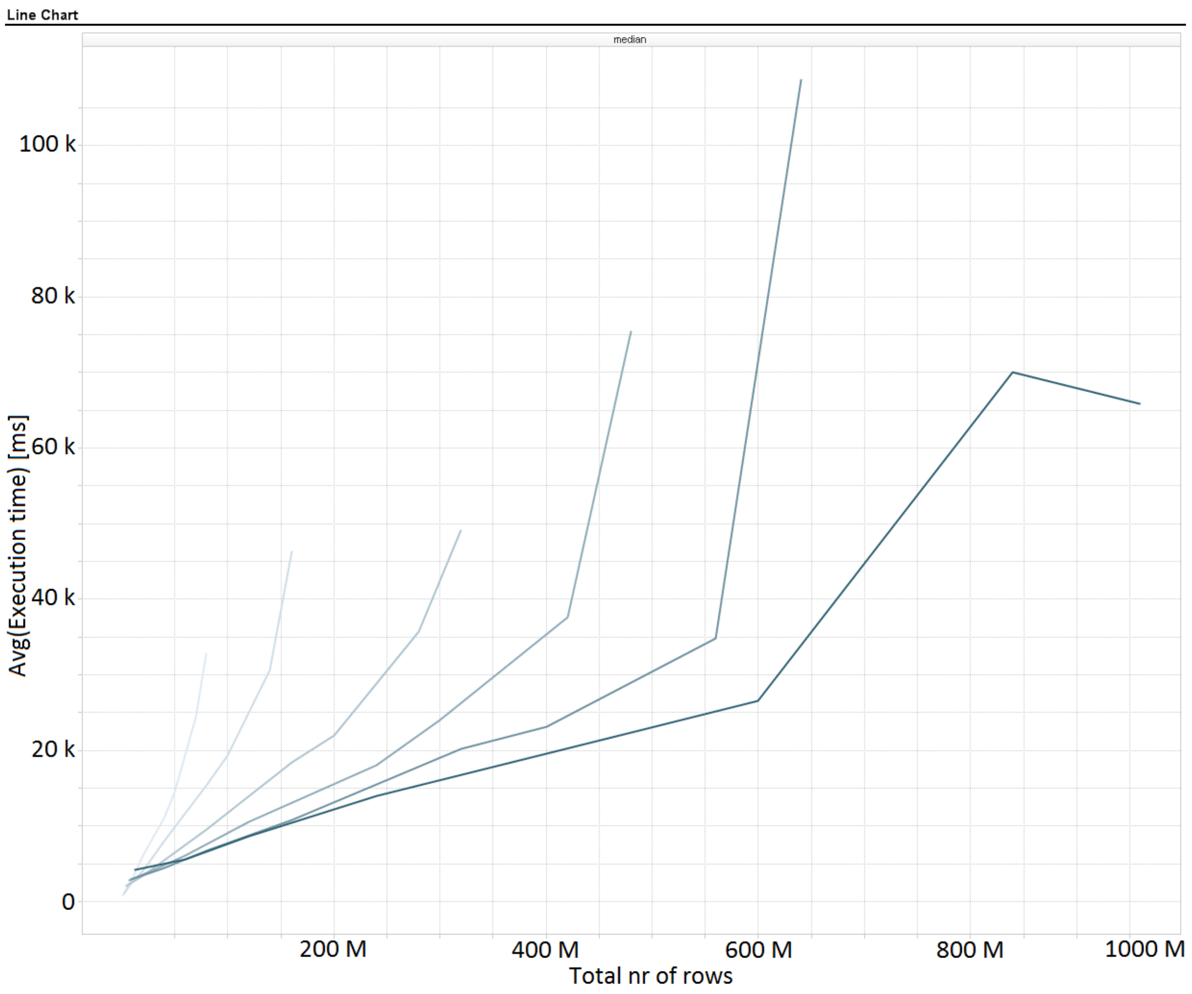

Figure 8: The median aggregation on Column 2 which consists of floating point values

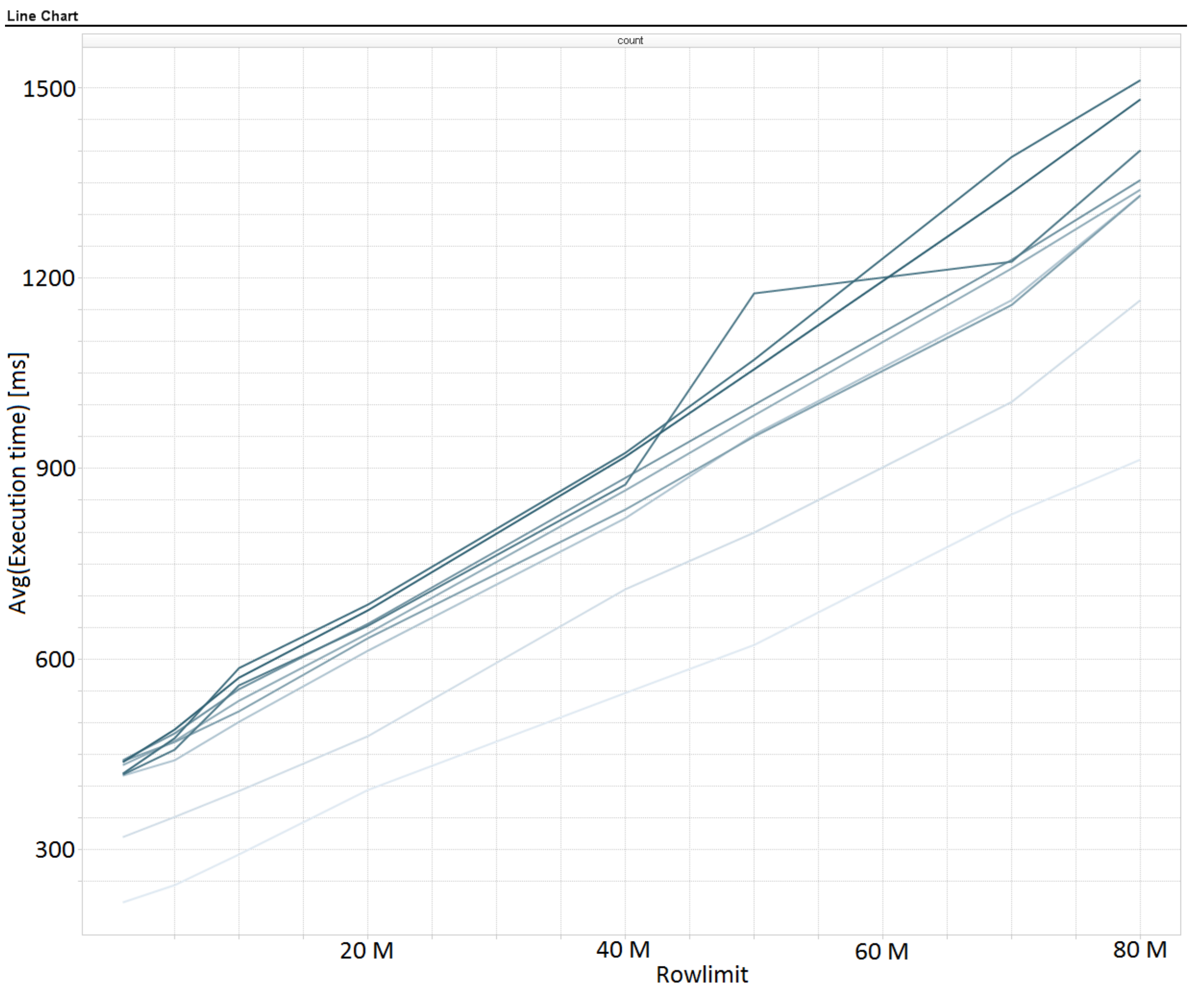

Figure 9: The response time of the count aggregation on Column 1 in relation to the number of rows per node

4.6 Why the results turned out the way they did

4.6.1 Unstable response times

The two main reasons for outstanding extreme values of the response time are initialization and memory shortage. Also, due to the probabilistic nature of the k-selection algorithm used to calculate the median, the response times on the median show a high variance. It should also be mentioned that all tests with more than eight nodes were conducted on a separate occasion, thus with a different set of computers. Because of this, eg. the response times for 12 nodes does not always fit in with the rest of the plot, but it is included to show our biggest tests.

Initialization

The table file is not read into the memory of the client until the first job is ordered. Therefore, the first job on any client will take a significantly longer time to perform than that of any following jobs of the same type. When we ran our tests, we loaded the server with job instructions in the afternoon, and we encouraged people to connect their clients when they went home that day. As a result of that, some clients connected well into that evening's tests and their first job always had an unproportionately long execution time.

The reason for this solution is that we wanted people to be able to have our client running before the tests actually started without it allocating unnecessary resources. This is because many tended to forget to start our client when they performed their customary go-home-routine, so we sent a reminding email earlier in the afternoon asking them to start our client.

Memory shortage
The Java Virtual Machine can only allocate so-much memory for its operations. The highest amount of memory that could be reserved was around 1600 MB, but that figure was not the same on different implementations of the JVM, different host operating systems, different architectures (such as 32 bit and 64 bit) and even different versions of the same implementation. As a result of this, some nodes were not able to hold all 80 million rows, which was the number of rows every node had locally, in memory. This limitation proved especially troublesome in our median calculation, where the same data was reread several times per calculation, yielding many page faults and poor performance. Therefore, the performance figures for the median calculations only show scalability reliably up to 60 million rows, after which the response time skyrockets.

In a production environment, this kind of problem would not exist on this scale, since an administrator would be able to control each node more precisely than what we were able to do on other employees' desktop computers, some of which also ran heavy background processes such as servers, virtual machines and debuggers.

The odd behavior of the 12 nodes case comes from the fact that those tests were run in a partially different environment. While still interesting due to the large size of data, at least one computer in that test was substantially slower,

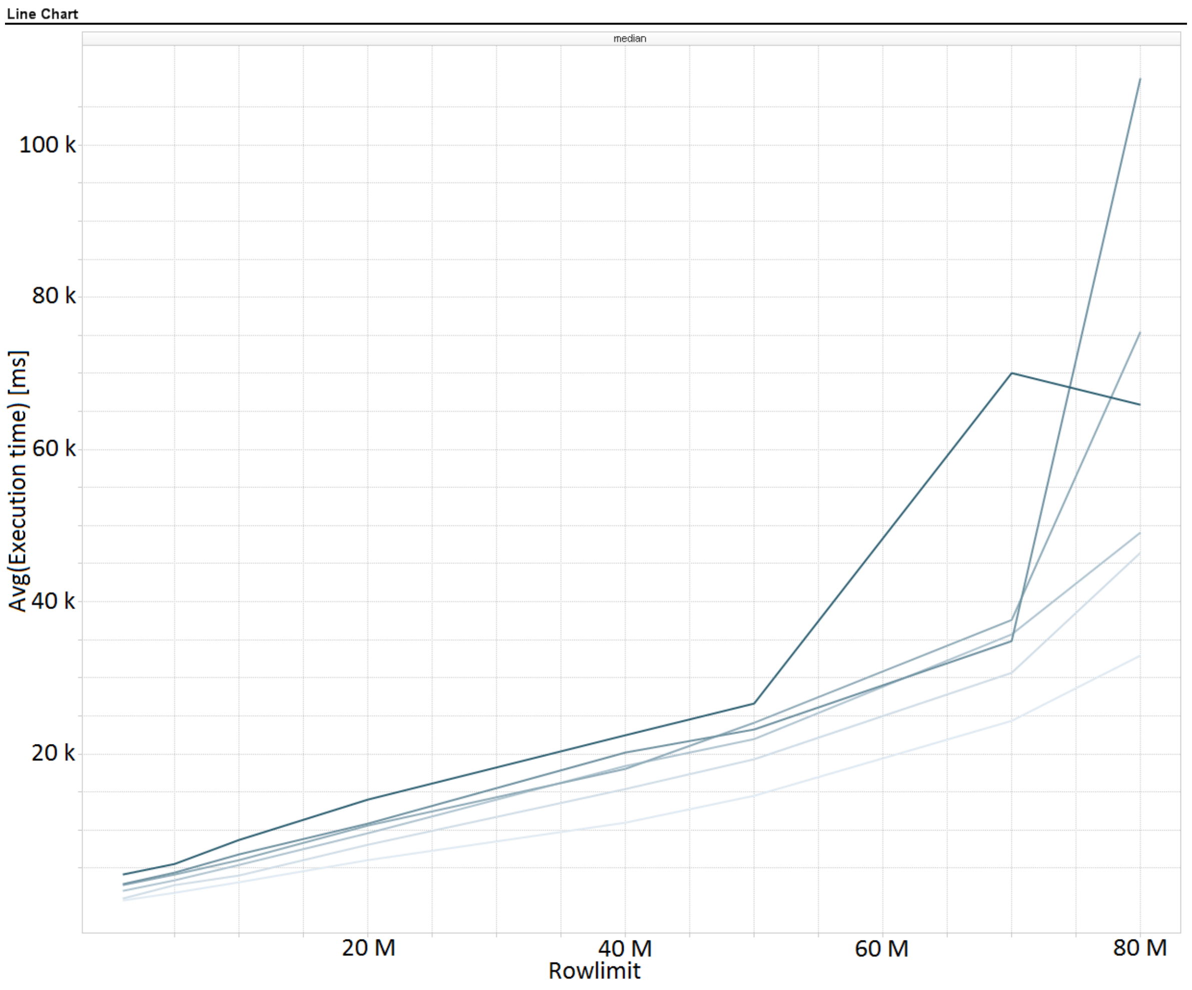

Figure 10: The average response time for the median aggregation climbs rapidly around 50-70M rows/node

probably due to memory shortage since performance on the other aggregations falls well into the pattern of fewer nodes. As visible in Figure 10, the continuing timeouts scew the results and only the ones fast enough get through.

k-selection randomness

The performance of most aggregations, measured by response time, excluding the median and with extreme values filtered out, was quite stable. The standard deviation is around 10%-20% of the average which means that one can predict the response time with quite high probability.

Due to the design of k-select 1, the response times from the median calculation spread a lot more. This is especially true in the case where the median of one of the integer columns was calculated. The values in the integer columns in our test ranged from 0 up to 14 and from 0 up to 19. This means that it is possible for the median calculation to finish after only one round, with a probability of 1/15 and 1/20 respectively. Thus, the standard deviation of calculating the median on an integer column is around 60% of the average time. This is a big difference, that means that it is hard to give a good estimate of how long time a median calculation is expected to take.

Calculating the average of a column with floating point numbers ranging uniformly between 0 and 1 is a lot more stable. This is because no longer, 1/15 or 1/20 of the values is a median, but instead only one value is the true median. While still substantially more unstable than the rest of the aggregations, with a standard deviation of around 30% of the average, the increased number of expected steps required to locate a particular element, makes calculating the median of a column with numerous unique values more stable than with only a few unique values.

4.6.2 Response time scalability

A clear trend in all of the visualizations of the response time of all aggregations is that as the number of rows increase, so does the response time. Also evident is that the response times seem to grow almost linearly with added data on a constant number of nodes, (except the median, which has a more than linear complexity and also ran into some problems as discussed in 4.6.1). Also, another clear trend is that with a growing number of nodes, the slope of the line of response times lowers. This is of course a great result as it shows that our algorithms scale well. As seen in Figure 11, the scaling is not linear, though. The performance gain by adding nodes decreases for every added node. The cause of this non-linear scalability comes partly from our server/client structure, but we have not tested any other structure, so we cannot claim for sure that eg. a tree structured network would suffer less from adding more nodes.

This shows that, with the size of data that is interesting for us, and for this particular set of problems, network traffic is not a limiting factor. In fact, an aggregation with 80 million lines on one node, has a response time very similar to an aggregation with 80 million lines per node on ten nodes. We have not precisely determined when it is no longer preferable to add another node, and thus splitting the data even more, but that ought not be very interesting since our prototype would not be used in production anyway.

4.7 Reliability

For a system like this to be usable in a production environment, reliability must be given a higher priority than what it was by us. Since our system existed primarily to get some real performance test results, reliability was given a low priority. To make sure that we would be able to perform a lot of tests, in excess of 100 000, the system had to stay up no matter what happened to the clients. It did not matter to our tests if data was lost when a client died since all data was created randomly. What really mattered was that the node came back up as soon as possible, and that the server would be able to perform additional tests on the other nodes in the meantime. The answer of the query is not what

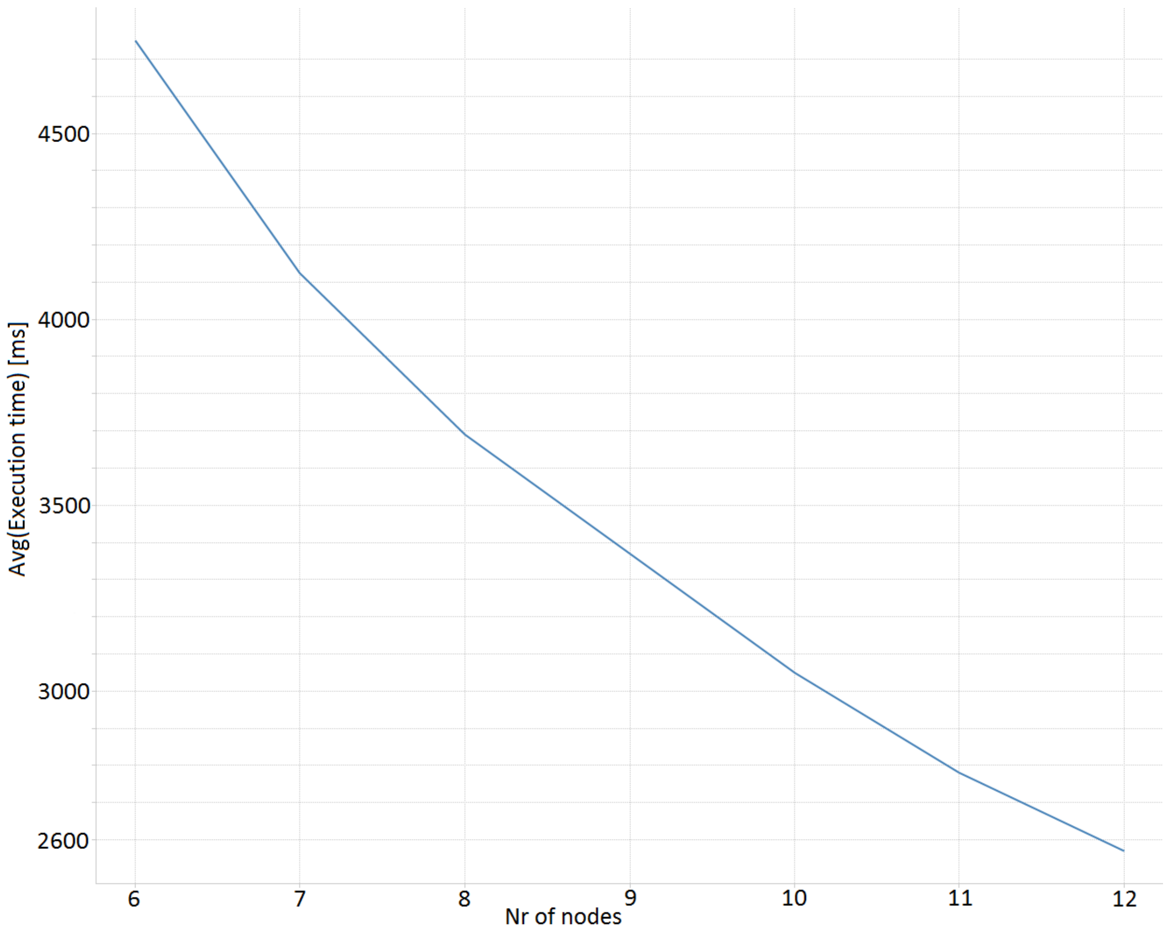

Figure 11: Sum 480 M total rows. Response time as a function of number of nodes with constant amount of data.

was interesting, instead it was how long time the query took, how many nodes that took part in the query and how many rows each node processed.

In a production environment, this behavior would of course not be acceptable. Every piece of data is valuable, and it is the answer that is important; the processing time is merely a nuisance. A production environment system would need to have data redundancy. While not in the focus of this report, there are several aspects to this matter, and there are several ways to implement data redundancy, where the implementation may also depend on the network topology. There is no definite answer to how many redundant copies there should be of every chunk, as it is a trade-off between risk (of a client actually failing), value (of the data), time (needed to spread the data) and resources (available storage space). It is probably safe to assume that value is not a factor here, since because of the relatively tedious task of transmitting the data to this system, all data has the same, high value. It is also reasonable to assume that if data is lost in this system, the data itself is not lost from its owner, since a backup ought to exist somewhere else. Lost data, in this context, only means that the distributed calculations system can no longer use that data to process queries.

Arguably, the biggest factor upon which is it decided how redundant data should be is the risk factor. In this context, risk is the probability that a client goes down, setting it to a state where it can no longer perform queries ordered by the root. Analyzing the possible reasons that might lead to a failure is important to properly conclude a correct risk factor.

Faults can be split into three categories, depending on how they occur.

- Transient fault A transient fault is a fault that only occurs rarely and irregularly, most often only once. The cause of the fault is not expected to reoccur and, given that the fault is handled correctly, the client should be able to remain operational. An example of a transient fault is a timeout caused by a slow start-up procedure of the client.
- Intermittent fault An intermittent fault is a fault that occurs regularly. The effects of this fault ought to be such, that if handled correctly, the client should be able to remain operational. However, the fault is expected to happen again. An example of an intermittent fault is a memory leak in the software, where a restart would temporarily redeem the problem.
- Permanent fault A permanent fault is a fault that causes the client to become unusable, where the node may not come back up without drastic measurements such as physically replacing the node or parts of it. A permanent fault may be caused by a hard disc crash or a broken network cable, to name a few.

There are many possible reasons to why a fault may occur. Here, they are divided into three categories.

- Hardware fault A hardware fault often causes a permanent fault, but it does not always have to be so. An obvious hardware fault which does cause a permanent fault is a hard disc crash, but is may also happen that a memory module becomes corrupt and returns incorrect data, which may lead to an intermittent fault.
- Software fault A software fault may be that the client software is erroneously programmed and contains bugs. Bugs may also cause all three sorts of faults, but they ought to be easier to predict. If, however, the software fault occurs somewhere else but in the client, such as the OS, the source of the fault may be hard to detect.
- Communication fault This category may very well fit into the other two categories, but since the context is distributed calculations, faults like Byzantine faults or simply a network overload are more easily put into this category.

One failure that goes a bit outside of the other failures is the value error. A value error means that the returned answer is not the correct one. This may be caused by either of the the reasons mentioned above, but to detect a value error, one possible solution is to reprocess the query. This may be done in parallel if the distributed calculations system is large enough; in other cases the query will simply have to be processed twice, at the cost of a doubled response time.

4.8 Environment of deployment

4.8.1 Programming language

The prototype system was built in Java. This was due to simplicity and platform independence; the client ran simultaneously on both Linux and Windows systems without any modifications. While performance in Java is acceptable for doing a performance analysis to show scalability, the real numbers of the execution times are not entirely representative for a final product. During development, we implemented parts of our operations in lower level languages such as C and C++, just to get a hint of how much performance is lost by using Java. Some operations, such as casting values from their byte representation on the file to integers and floating point variables took practically no time at all in C, while those operations in Java were quite costly. This operation, however, can be considered a one-time operation and ought not affect the operational performance.

However, also most calculations were faster in C/C++ compared to Java. Simple operations such as iterating through an array and calculating the sum of the elements were indeed faster than in Java, but the difference was not big enough to motivate that we should use one of these languages to develop the whole system, with features such as multi-threading, reliability and other things that was not the focus of this report.

On operations such as a join, local execution time has a lesser effect on overall performance than on simple aggregations such as the sum. These simple aggregations already have acceptable performance, so one might consider that they are not where effort to improve performance should be put. Especially when materializing a join the bottleneck is likely to lie not in the performance of the software, but rather on the capacity of the network.

A system built in a higher level language also has the advantage of being easier to deploy in a mixed environment, where clients run different operating systems on different hardware. Much low level code, such as code for synchronization and communication will have to be written specifically for each operating system. This leads to a higher cost of development, a bigger risk of bugs entering the system and less maintainable code, since more in depth knowledge of each particular platform is required. If written and tuned correctly, however, the system will probably perform better.

4.8.2 Dedicated worker park vs unused capacity leeching

A distributed system for calculations can be deployed on either a dedicated (to any given degree of dedicated) server park, as a background process on desktop computers, or as a mix thereof.

If each partial calculation is performed at one node only, most aggregations will have the response time of the slowest node. Because of this, it is a bad idea to use office desktop computers as the primary source of hosts for nodes in the system. Office desktop computers may experience extreme transient declines in performance, originating from tasks such as compilations, debugging, anti-virus scans etc. A server park with dedicated computers is a better source for hosts. A server in a dedicated park provides a more deterministic behavior compared to a solution using office desktop computers.

Office desktop computers may act as an accelerator, though. It is a possible scenario that office desktop computers are given much fewer chunks than their dedicated counterparts. They may also share chunks with more computers, so that each job may be performed at several locations at the same time. This kind of system can improve performance without the need of extra hardware.System architecture The characteristics of a complete distributed system for calculations, as described in this thesis, are similar to that of a distributed read-only database, somehow connected to a data source. This data source might be just a huge text file, or it might be some sort of a database, where for example a join is materialized.

4.9 Further work

Here, we present what we think would be interesting areas to conduct further work in the area of distributed calculations towards a visualization tool.

4.9.1 Distributing visualization jobs

The response of jobs such as clusterings may be very large, as it may need to identify single elements that should belong to specific clusters. This in turn may result in long response times, since apart from the actual aggregation, also returning the answer may take a considerable amount of time.

One possible solution to this is to distribute the actual visualization job. Instead of returning a map structure where every element is mapped onto one cluster, the response could be some sort of bitmap representing the actual visualization. Given a resolution of the visualization, such a solution would mean that the size of the response would not grow with the size of the data.

4.9.2 More users vs. short response time

If many users want to work on the same data set at the same time, then computation jobs would simply have to be queued (and perhaps "rescheduled").

The case where several users want to work with different data sets at the same time is more troublesome. It is likely that there will not be enough main memory to hold all simultaneously data sets, resulting in a lot of performance loss due to the switching needed to fetch in another data set every time a user sends a query. The in-memory structure would need to be reexamined and it is likely that one would instead want the data to reside on disk, and instead use a RAID configuration that makes the disk bandwith increase. By doing this, the big context switch will be eliminated in favor for a more stable, although likely slower, computation time.

Using the disk as primary storage opens many doors for further improvements, like implementing load-balancing into the system. This becomes possible since any node that has a particular chunk of interest on the disk can perform a query, making it possible to migrate parts of jobs.

4.9.3 Indexing

Since the prototype was built around an in-memory architecture, the area of indexes was not deeply investigated [16]. If a disk-based architecture is to be examined, indexing would need to be investigated more thoroughly. Problems include things such as knowing what columns to index, perhaps also using an algorithm that concludes what columns that are used most frequently.

5 Conclusion

In this report we have investigated the use of a distributed system as a back-end query system for use in a visualization tool. We have implemented a computation system in a server/client fashion in which different problems and algorithms has been tested. Overall, distribution of the relevant algorithms has shown promising results.

The use of Hadoop as a back-end system was analyzed and deemed infeasible. The underlying abstraction is not ill fit in itself, but Hadoop is built with other goals in mind than low response times, which is a crucial thing for a visualization tool. Also, the abstraction makes algorithms demanding high control over communication harder to implement.

By using the knowledge of data, and that it will always follow a certain structure, a tailor made system might outperform a more general approach. The structure and usage pattern for a visualization back-end is close to that of a database, since it shares many data structures and type of queries.

A read-only distributed database-like system for calculations, without the overhead needed for transactions and rebuilding of indexes is probably the closest fit for the needs. A heavily simplified prototype of such as system was built and tested to see what response times that was to be expected.

To our understanding, the prototype described here is fast enough for most operations required by a visualization tool, perhaps with the exclusion of the median which took a fair amount of time. Additional algorithms like clustering algorithms appears to yield promising result when distributing, but are not tested.

Examining distributed databases, and their aggregation speed, in comparison to a tailor made distributed system would be an interesting continuation of the work described here, as we believe that it is not clear exactly what kind of system would be best fit.

References

[1] Ieee standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985*, pages –, Aug 1985.

[2] MySQL AB. Mysql :: Mysql 6.0 reference manual :: 7.4.4 how mysql uses indexes. `http://dev.mysql.com/doc/refman/6.0/en/mysql-indexes.html`, 2009.

[3] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980, New York, NY, USA, 2008. ACM.

[4] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[5] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. Np-hardness of euclidean sum-of-squares clustering. *Mach. Learn.*, 75(2):245–248, 2009.

[6] David A. Bader. An improved, randomized algorithm for parallel selection with an experimental study. *J. Parallel Distrib. Comput.*, 64(9):1051–1059, 2004.

[7] Lawrence Livermore National Laboratory Blaise Barney. Message passing interface (mpi). `https://computing.llnl.gov/tutorials/mpi/`, 2009.

[8] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Linear time bounds for median computations. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 119–124, New York, NY, USA, 1972. ACM.

[9] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.

[10] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *FIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264, New York, NY, USA, 1974. ACM Press.

[11] Inc. Cloudera. Thinking at scale. `http://www.cloudera.com/hadoop-training-thinking-at-scale`, 2009.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[13] Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Revised Papers from Large-Scale Parallel Data Mining, Workshop on Large-Scale Parallel KDD Systems, SIGKDD*, pages 245–260, London, UK, 2000. Springer-Verlag.

[14] The Apache Software Foundation. Hive - hadoop wiki. `http://wiki.apache.org/hadoop/Hive`, 2009.

[15] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.

[16] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):509–516, 1992.

[17] Constance L. Hays. What wal-mart knows about customers' habits. `http://www.nytimes.com/2004/11/14/business/yourmoney/14wal.html?pagewanted=print`, 2004.

[18] ISO/IEC. Information technology – database languages – sql – part 13: Sql routines and types using the java tm programming language (sql/jrt). Technical report, International Organization for Standardization, Geneva, Switzerland., 17. July 2008.

[19] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[20] Main Memory Jun, Jun Rao, and Kenneth A. Ross. Cache conscious indexing for decision-support in. pages 78–89, 1999.

[21] Fabian Kuhn, Thomas Locher, and Roger Wattenhofer. Tight bounds for distributed selection. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 145–153, New York, NY, USA, 2007. ACM.

[22] Fabian Kuhn, Thomas Locher, and Roger Wattenhofer. Distributed selection: a missing piece of data aggregation. *Commun. ACM*, 51(9):93–99, 2008.

[23] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002.

[24] The Message Passing Interface Forum (MPIF). Mpi: A message-passing interface standard. `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`, 1995.

[25] Inc. Owen O'Malley, Yahoo! Grid Computing TeamSun Microsystems. Apache hadoop wins terabyte sort benchmark (hadoop and distributed computing at yahoo!). `http://developer.yahoo.net/blogs/hadoop/2008/07/apache_hadoop_wins_terabyte_sort_benchmark.html`, 2008.

[26] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178, New York, NY, USA, 2009. ACM.

[27] Nigel Pendse. The olap report: What is olap? `http://www.olapreport.com/fasmi.htm`, 2005.

[28] Liuba Shrira, Nissim Francez, and Michael Rodeh. Distributed k-selection: From a sequential to a distributed algorithm. In *PODC '83: Proceedings of*

the second annual ACM symposium on Principles of distributed computing, pages 143–153, New York, NY, USA, 1983. ACM.

[29] SMHI. Arets vader. `http://www.smhi.se/cmp/jsp/polopoly.jsp?d=10989&l=sv`, 2009.

[30] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented dbms. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[31] Inc. Sun Microsystems. Hashmap (java platform se 6). `http://java.sun.com/javase/6/docs/api/index.html?java/util/HashMap.html`, 2008.

[32] Inc. Sun Microsystems. Developer resources for java technology. `http://java.sun.com/`, 2009.

[33] J. Villadangos, A. Cordoba, F. Farina, and M. Prieto. Efficient leader election in complete networks. In *PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 136–143, Washington, DC, USA, 2005. IEEE Computer Society.

A Performance graphs

A.1 Average

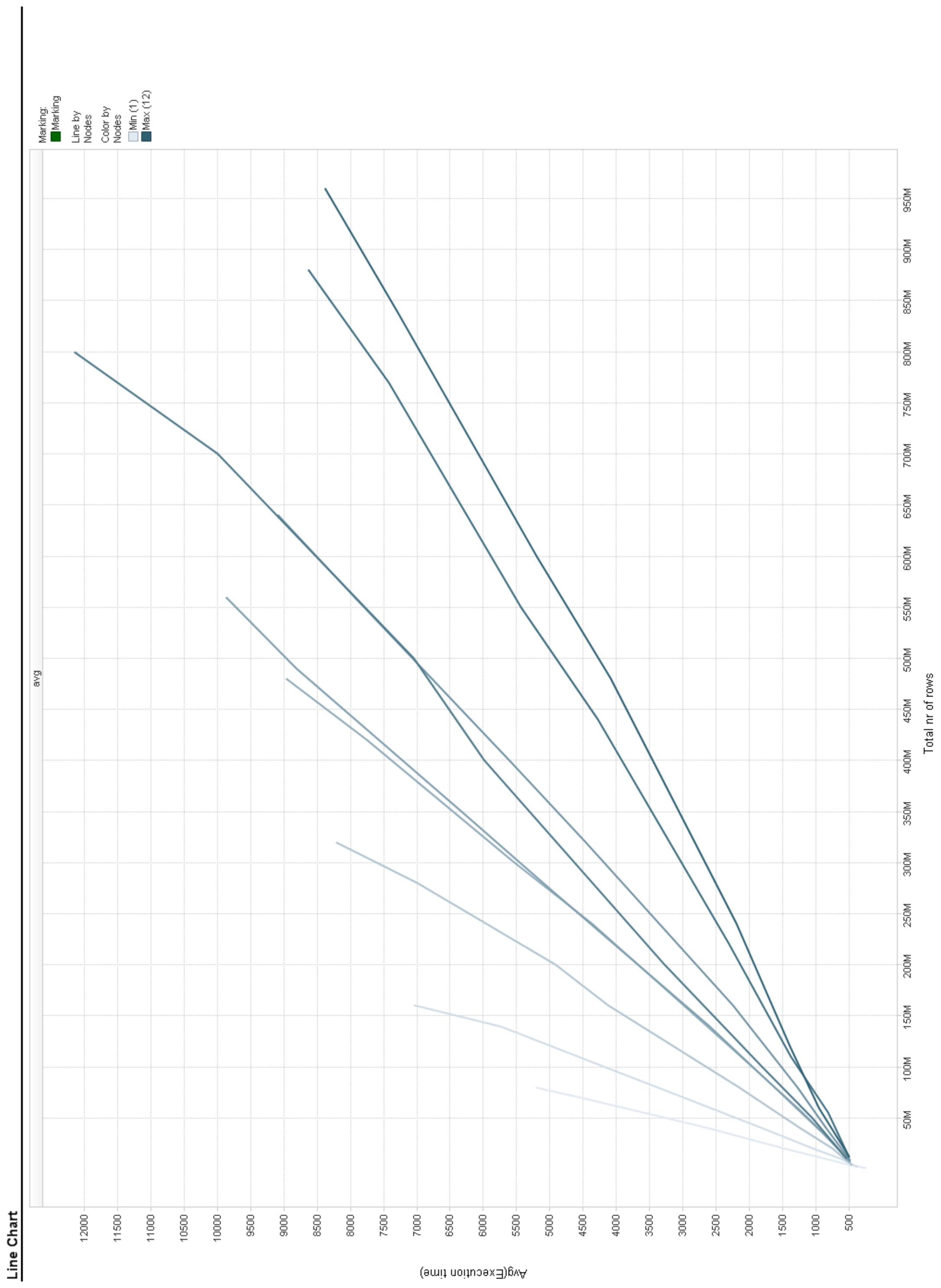

Figure 12: Average response time on column 1 per total nr of rows

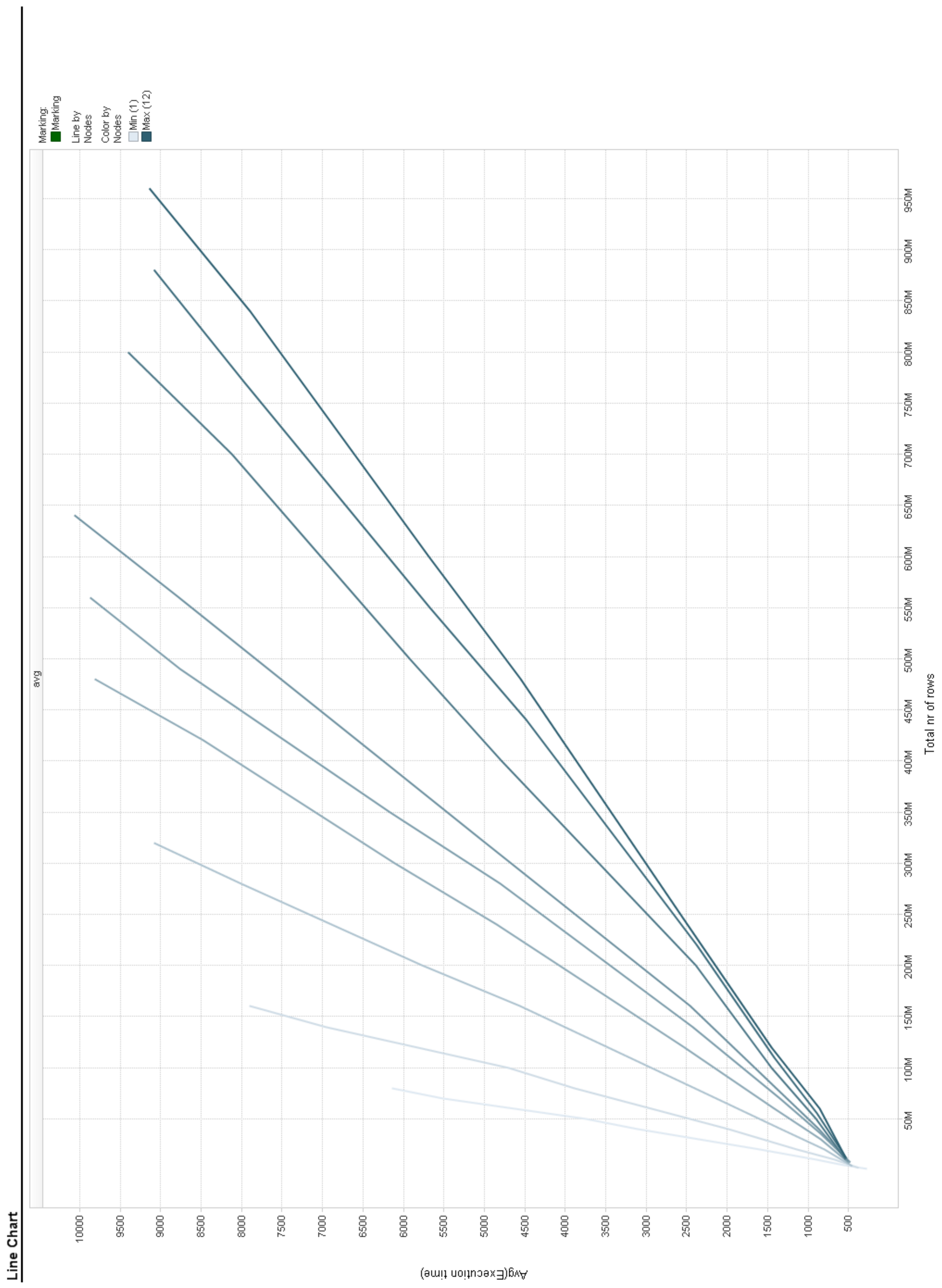

Figure 13: Average response time on column 2 per total nr of rows

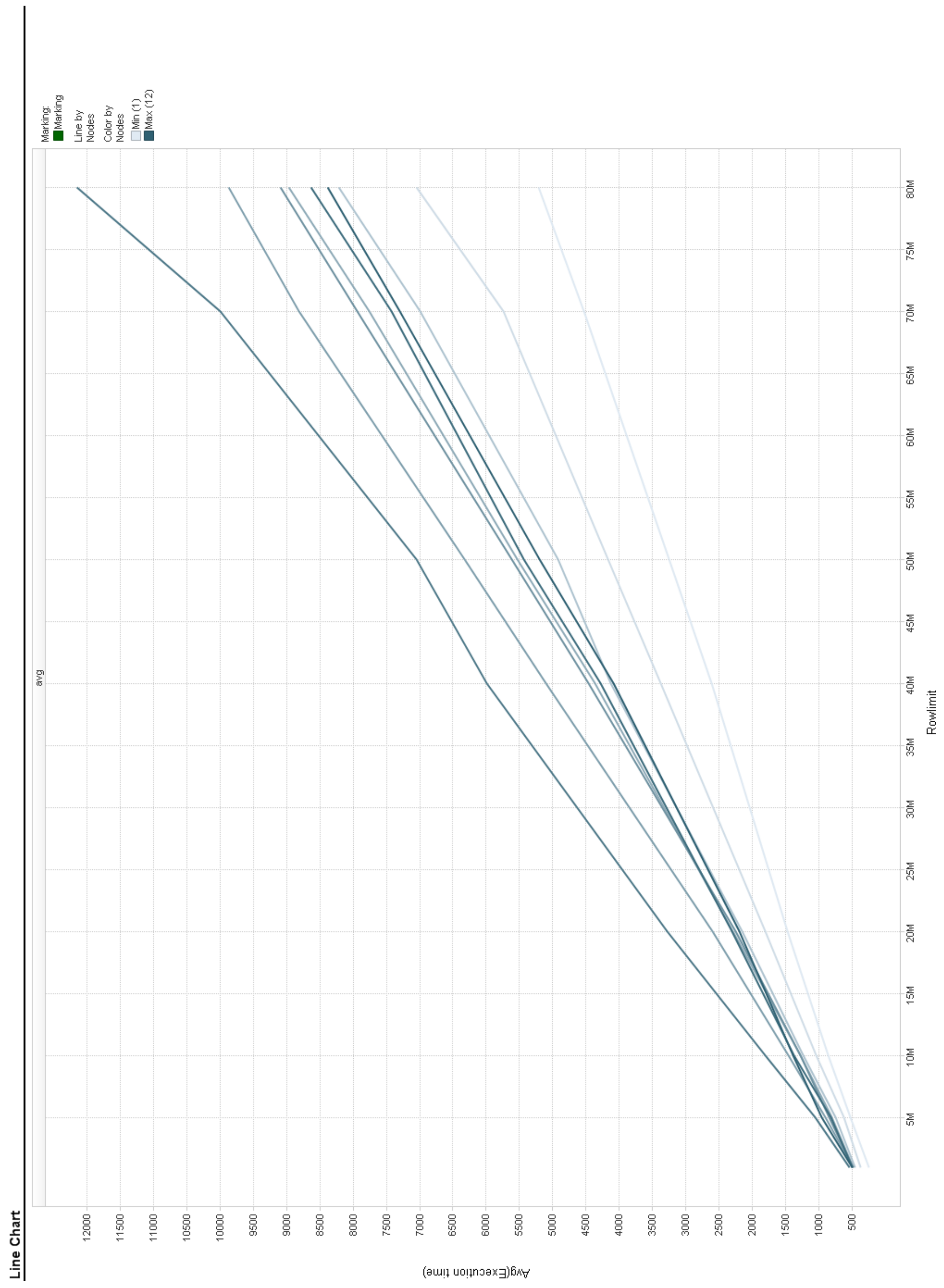

Figure 14: Average response time on column 1 per rows / node

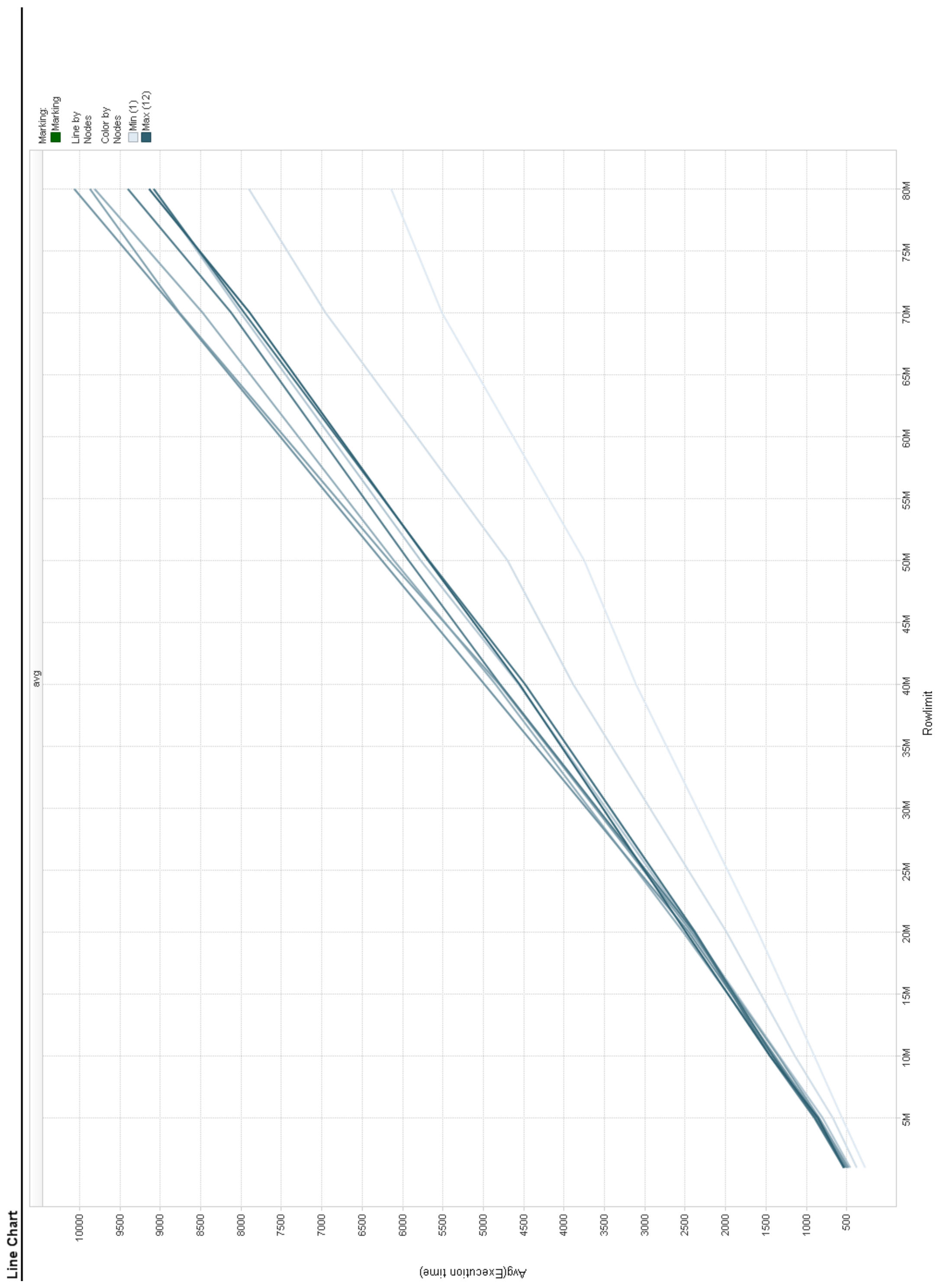

Figure 15: Average response time on column 1 per rows / node

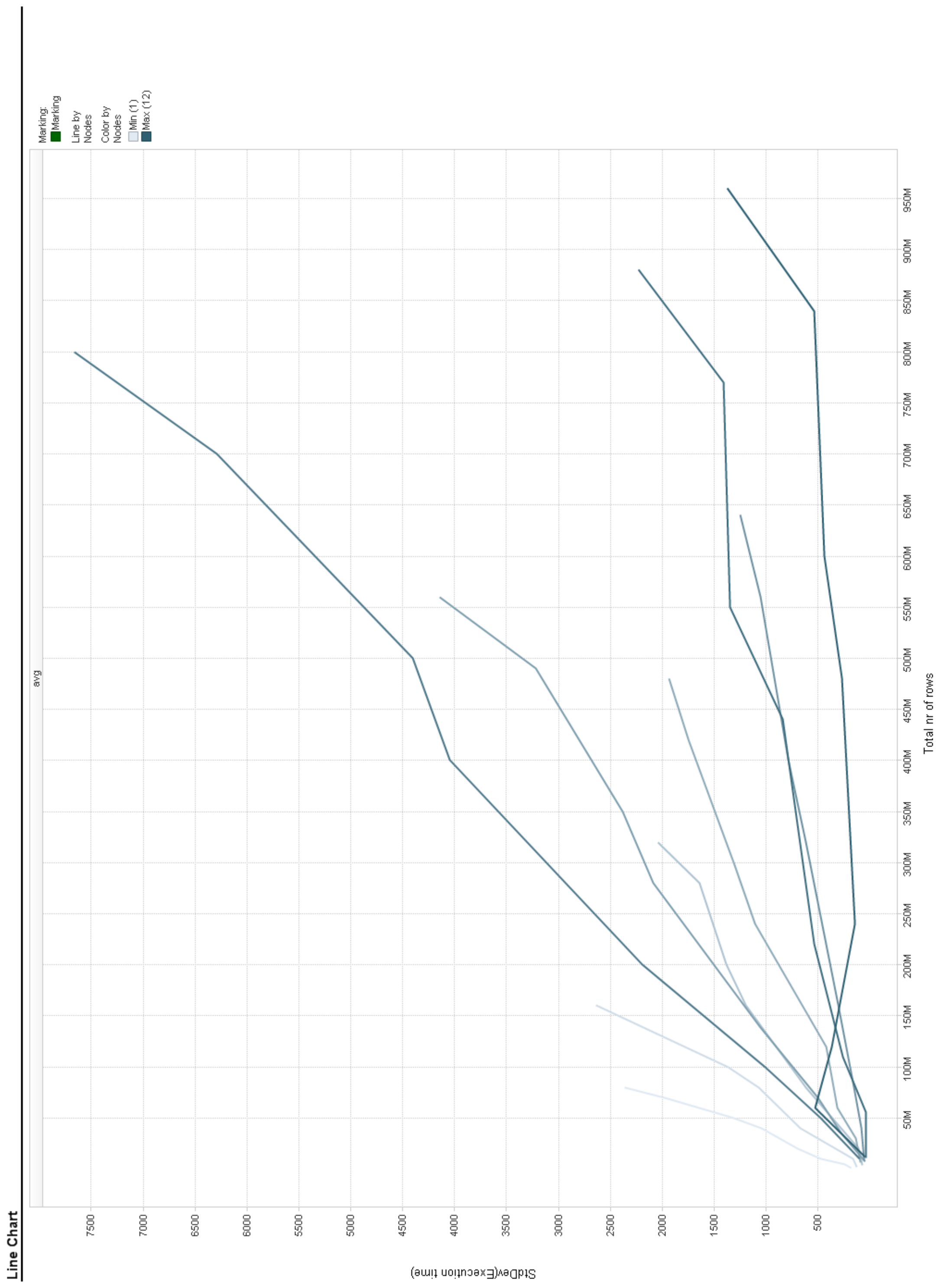

Figure 16: Standard deviation on column 1 per total nr of rows

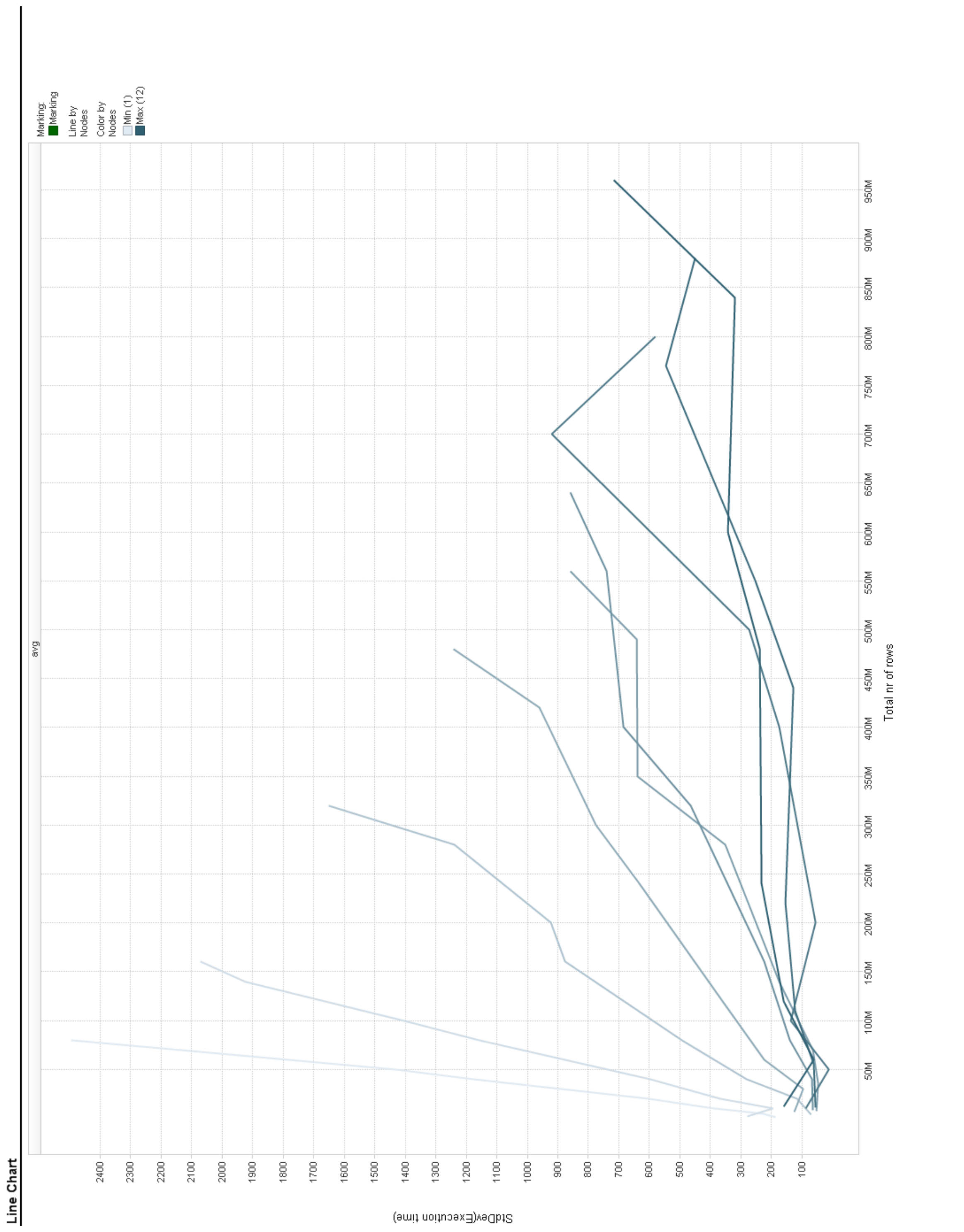

Figure 17: Standard deviaton on column 2 per total nr of rows

A.2 Count

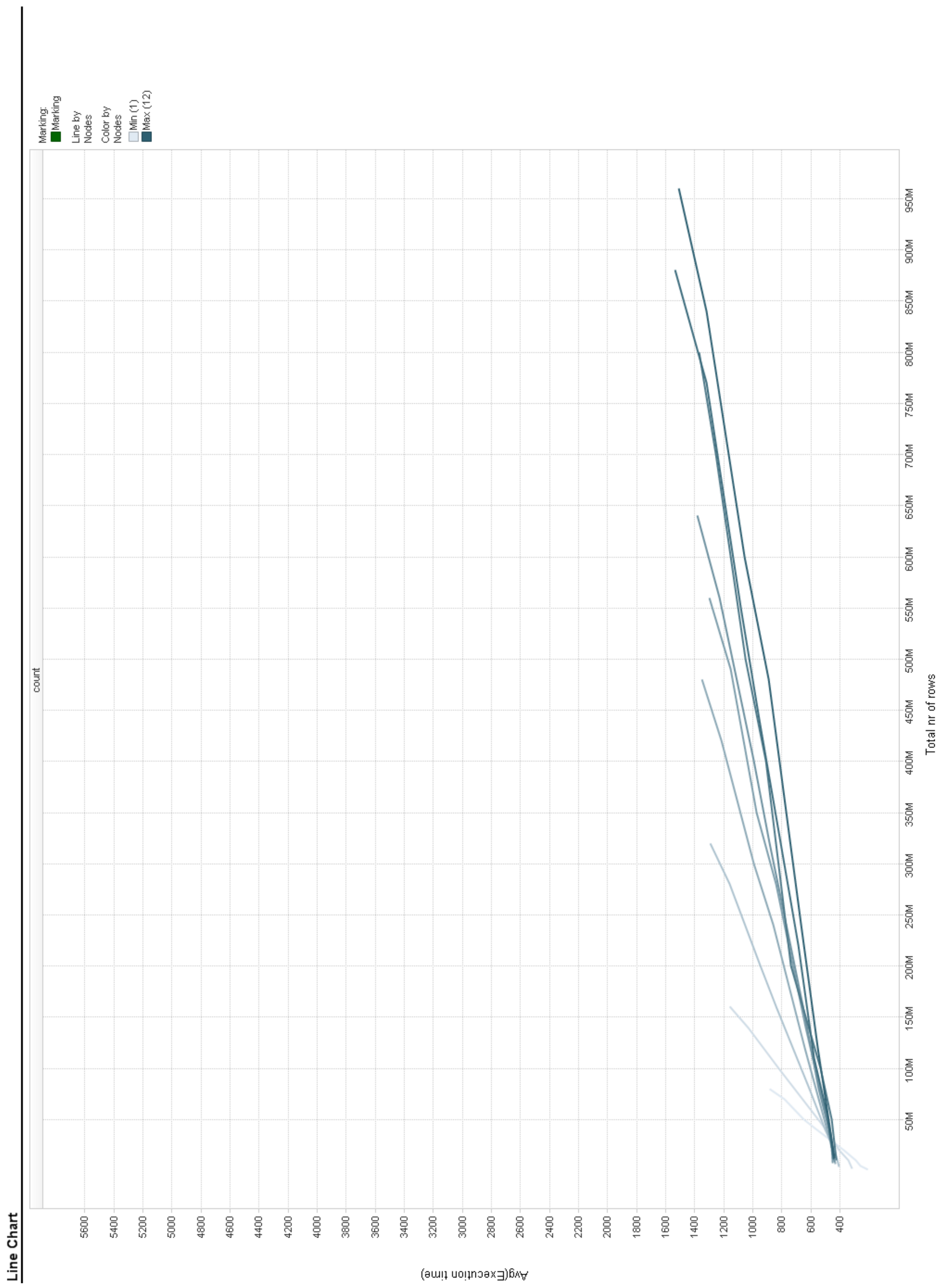

Figure 18: Average response time on column 1 per total nr of rows

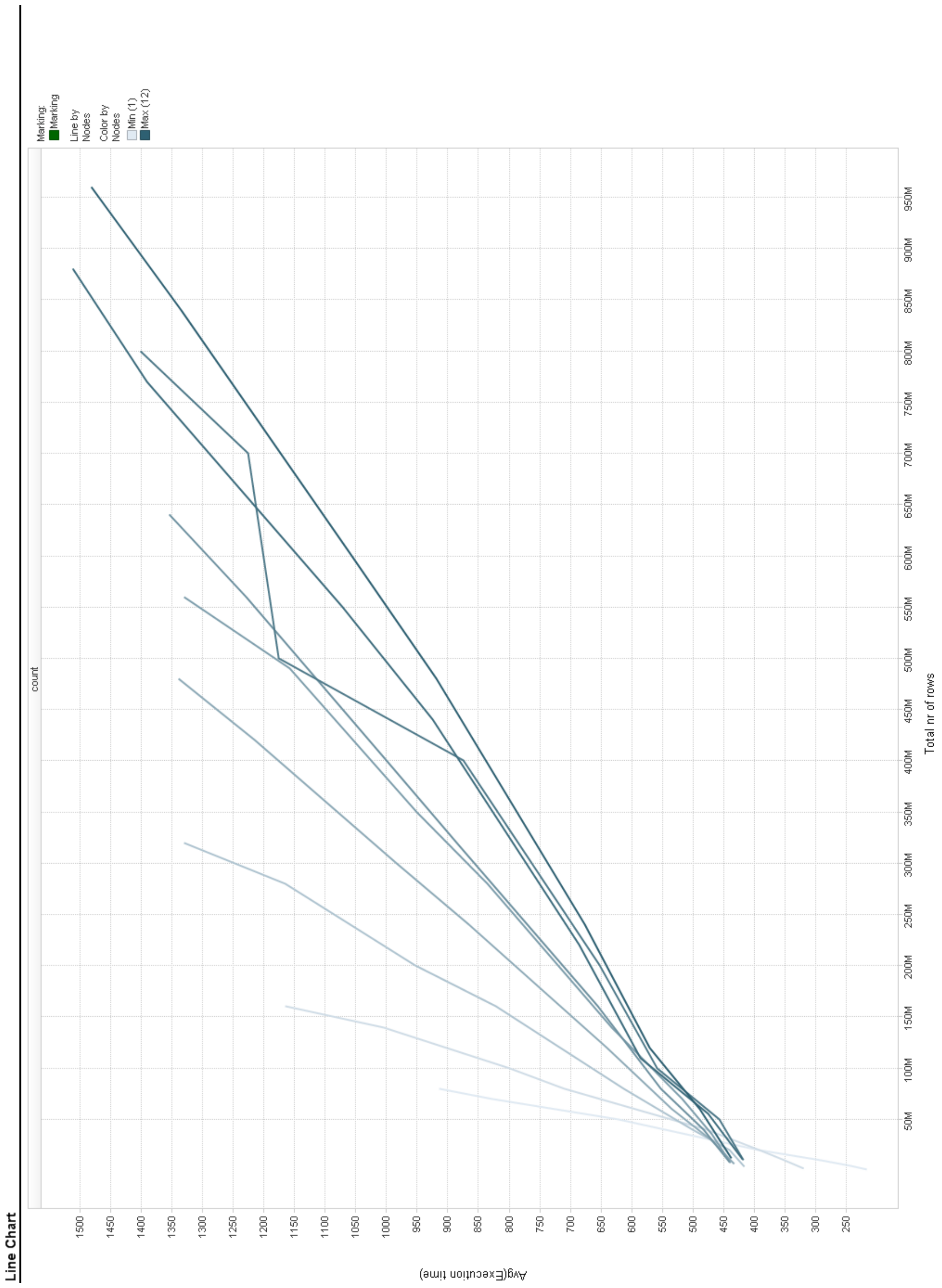

Figure 19: Average response time on column 2 per total nr of rows

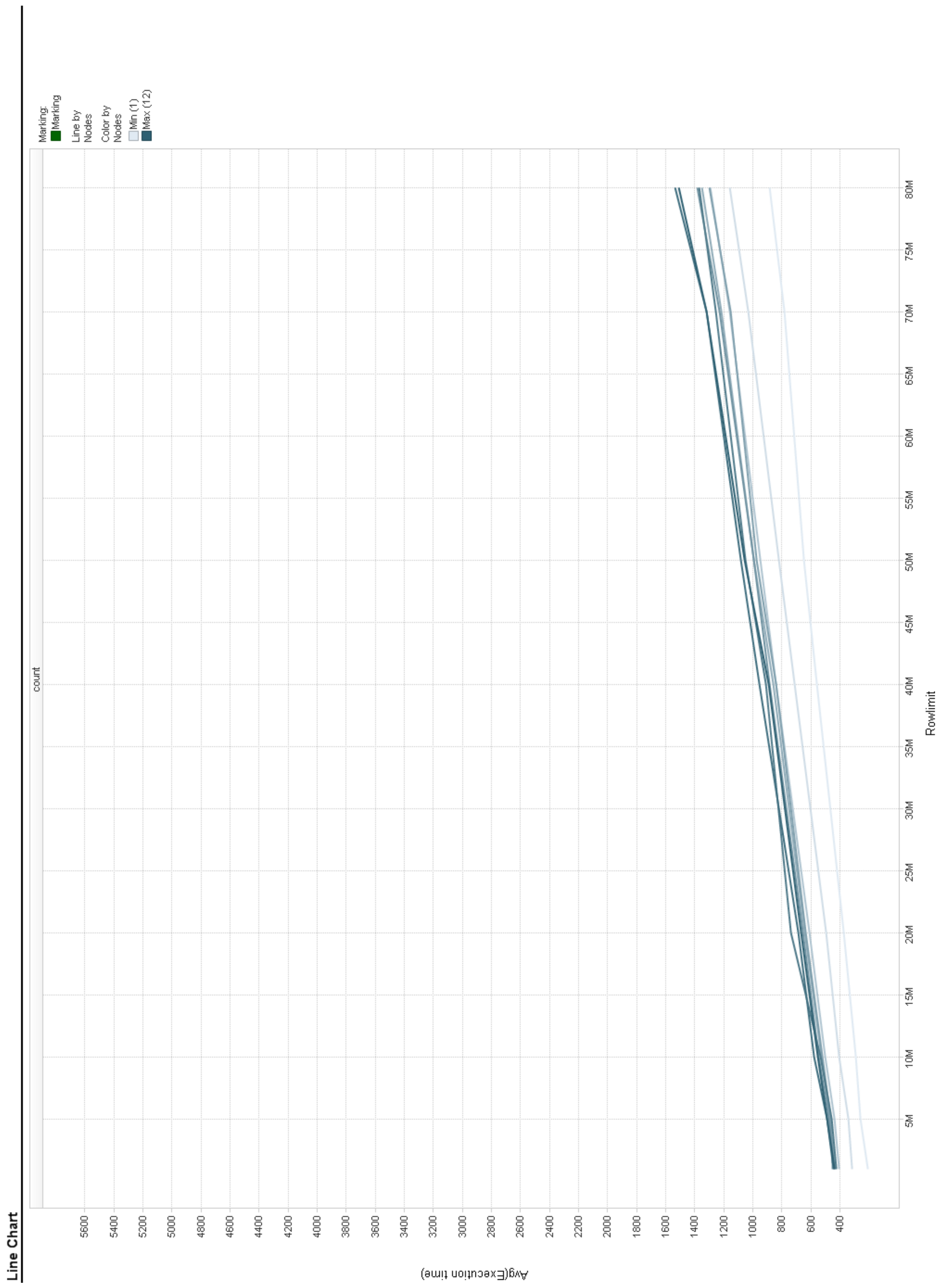

Figure 20: Average response time on column 1 per rows / node

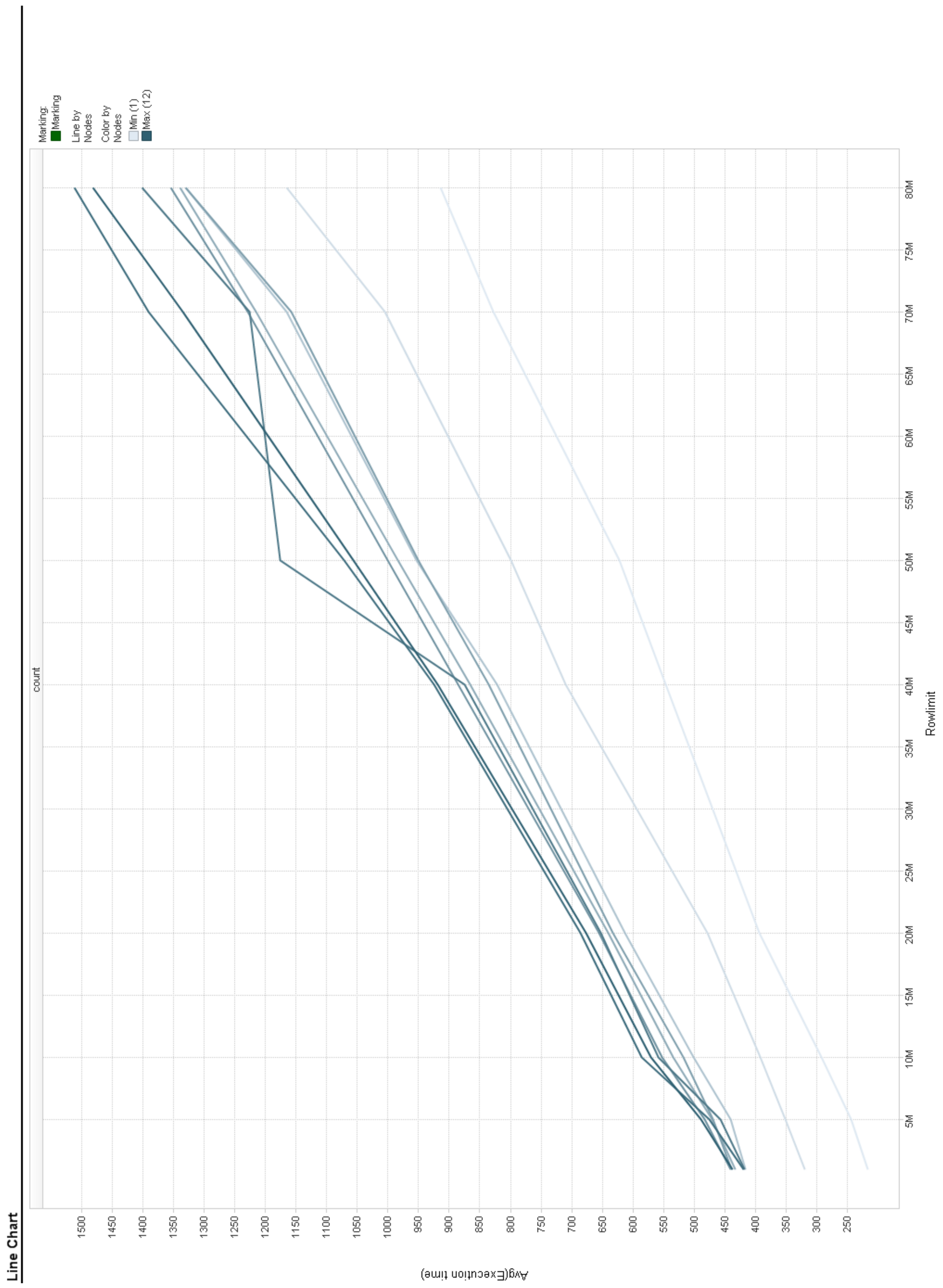

Figure 21: Average response time on column 1 per rows / node

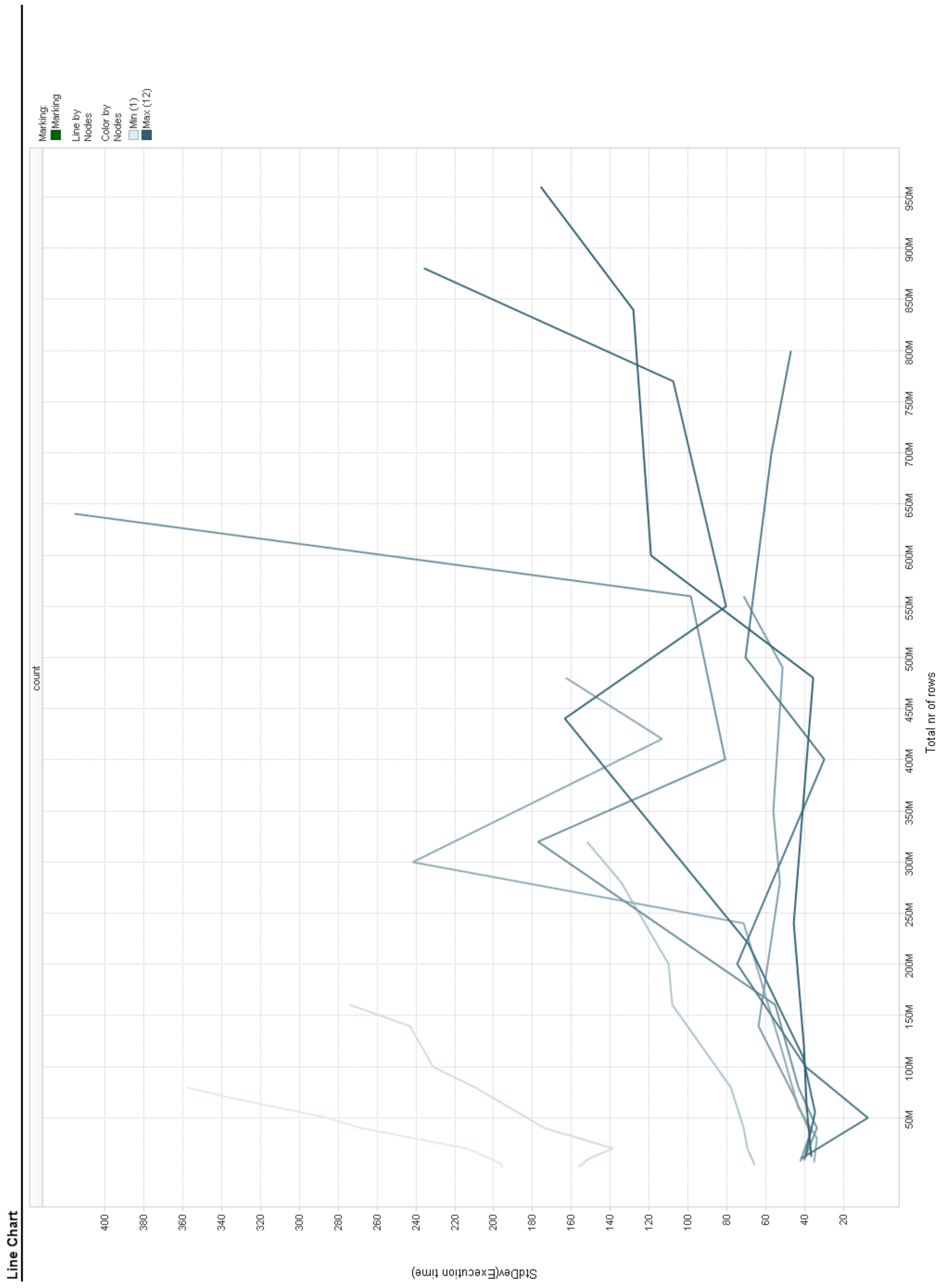

Figure 22: Standard deviation on column 1 per total nr of rows

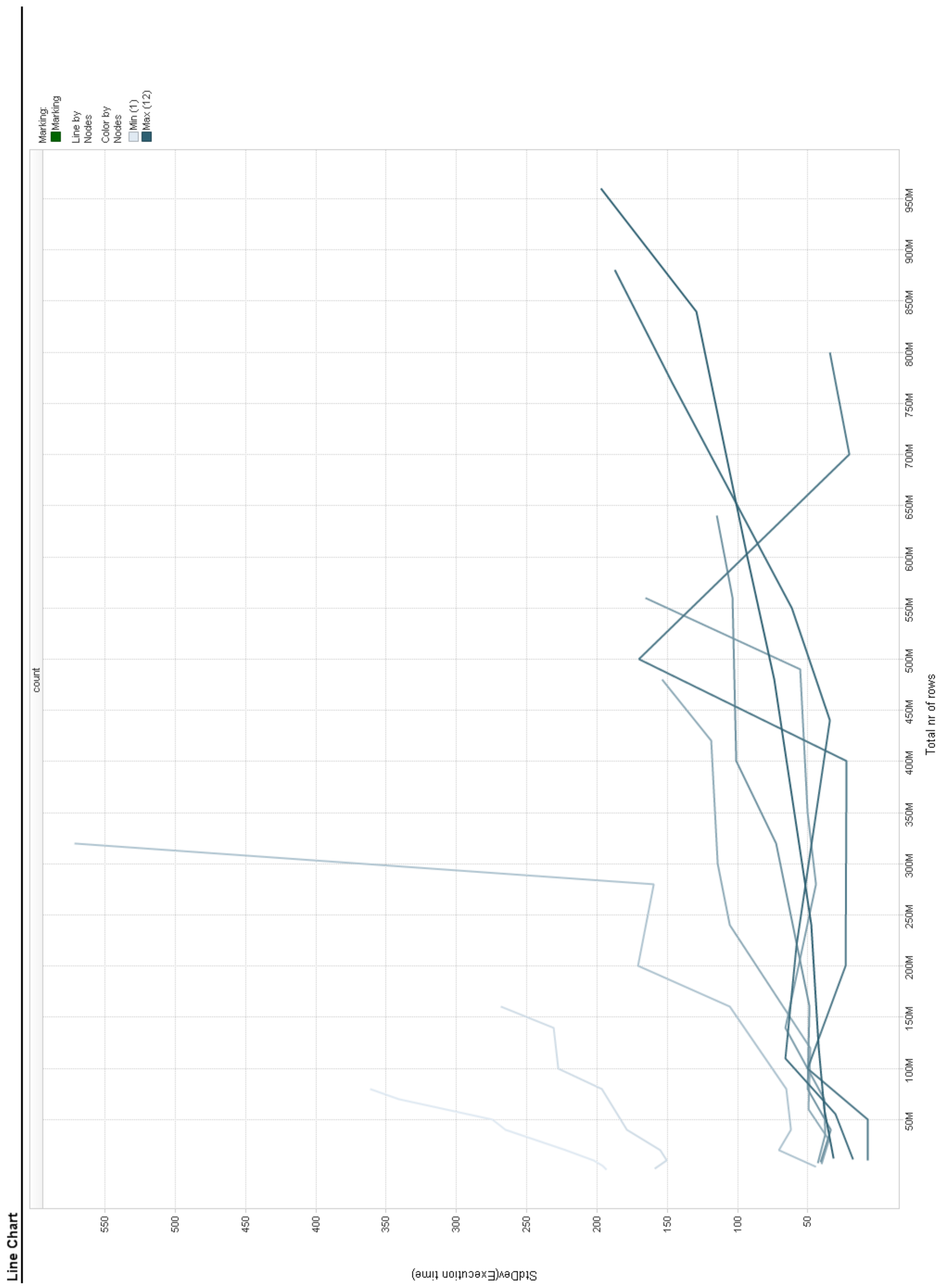

Figure 23: Standard deviaton on column 2 per total nr of rows

A.3 Max

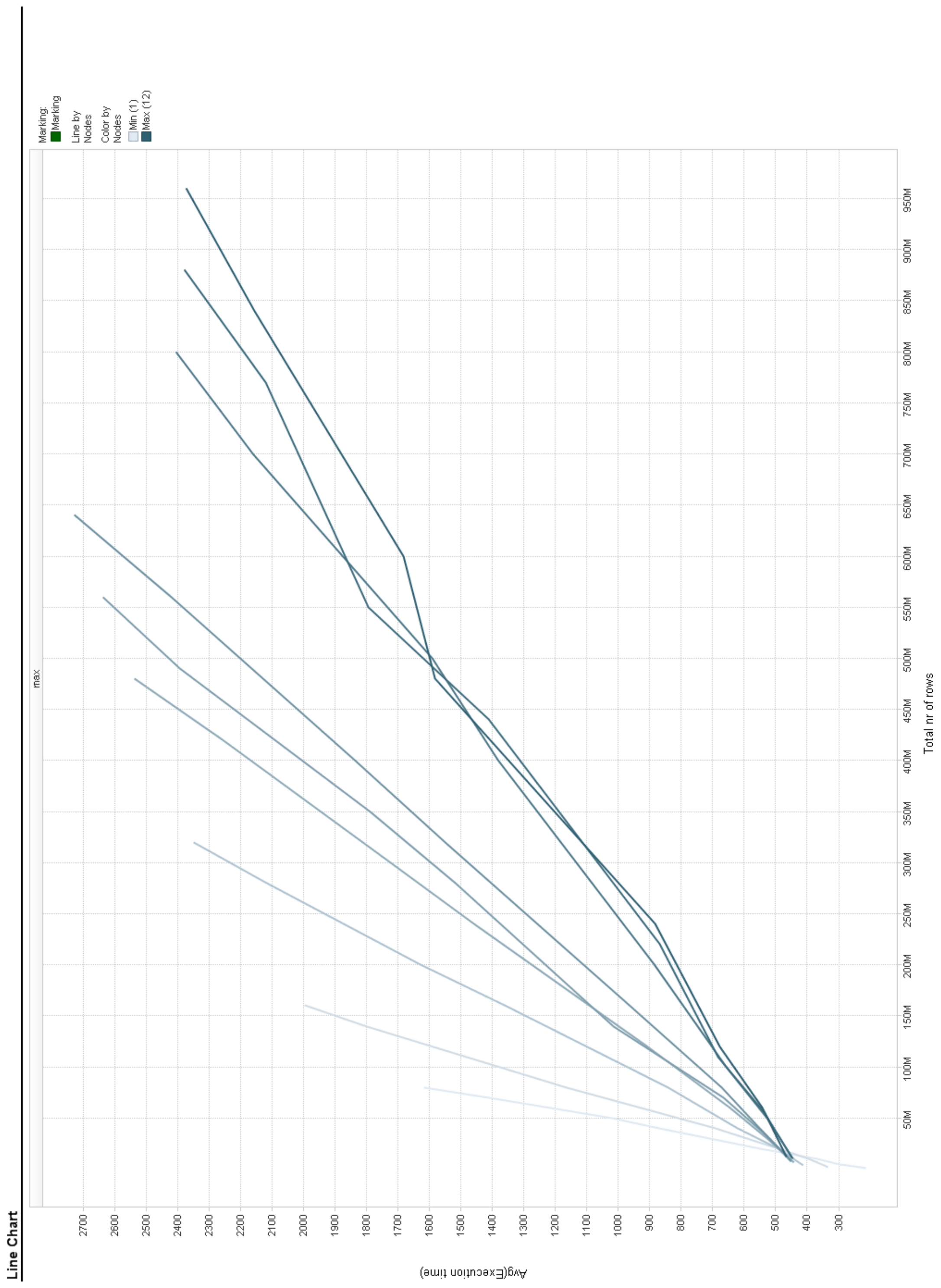

Figure 24: Average response time on column 1 per total nr of rows

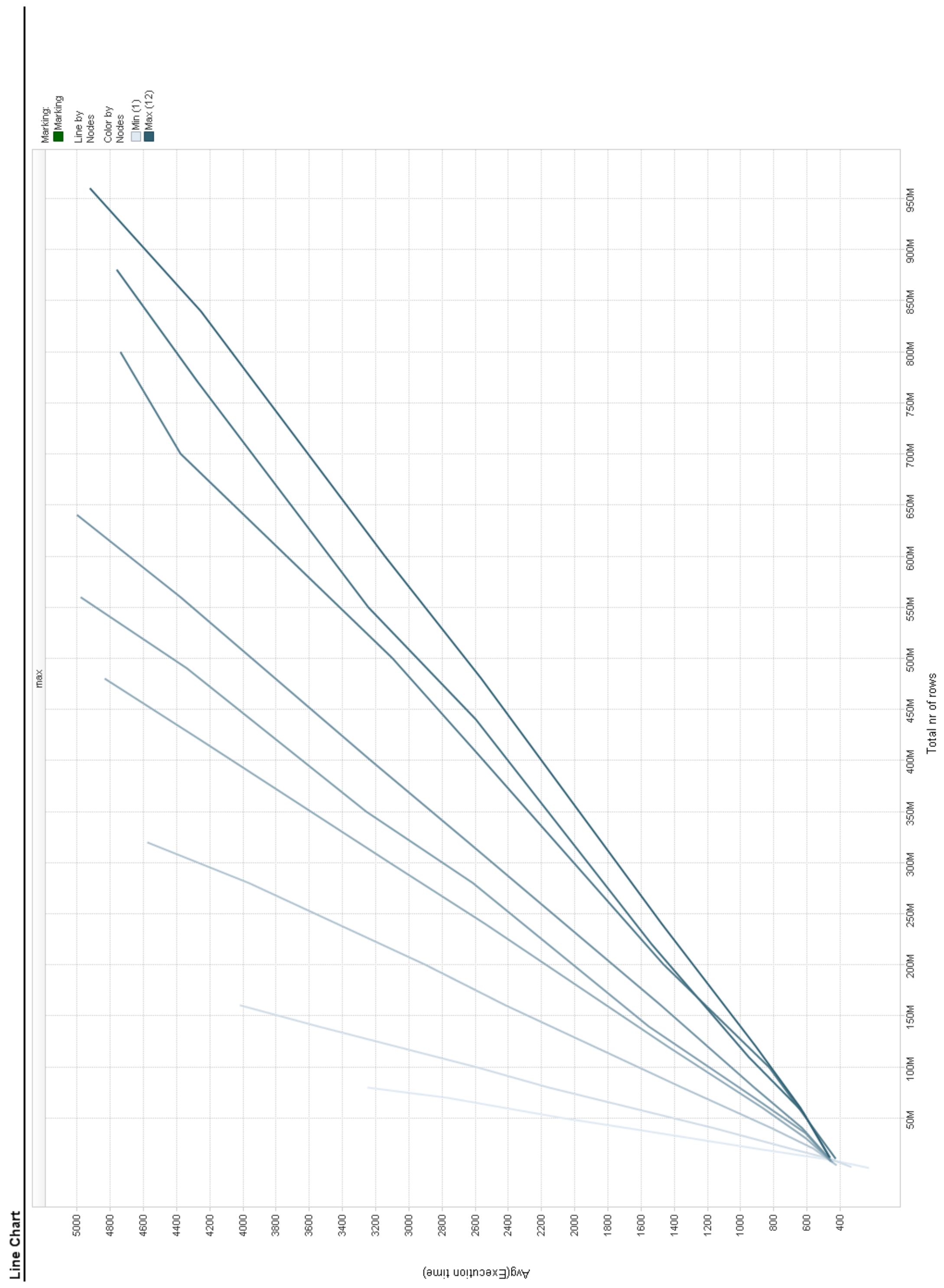

Figure 25: Average response time on column 2 per total nr of rows

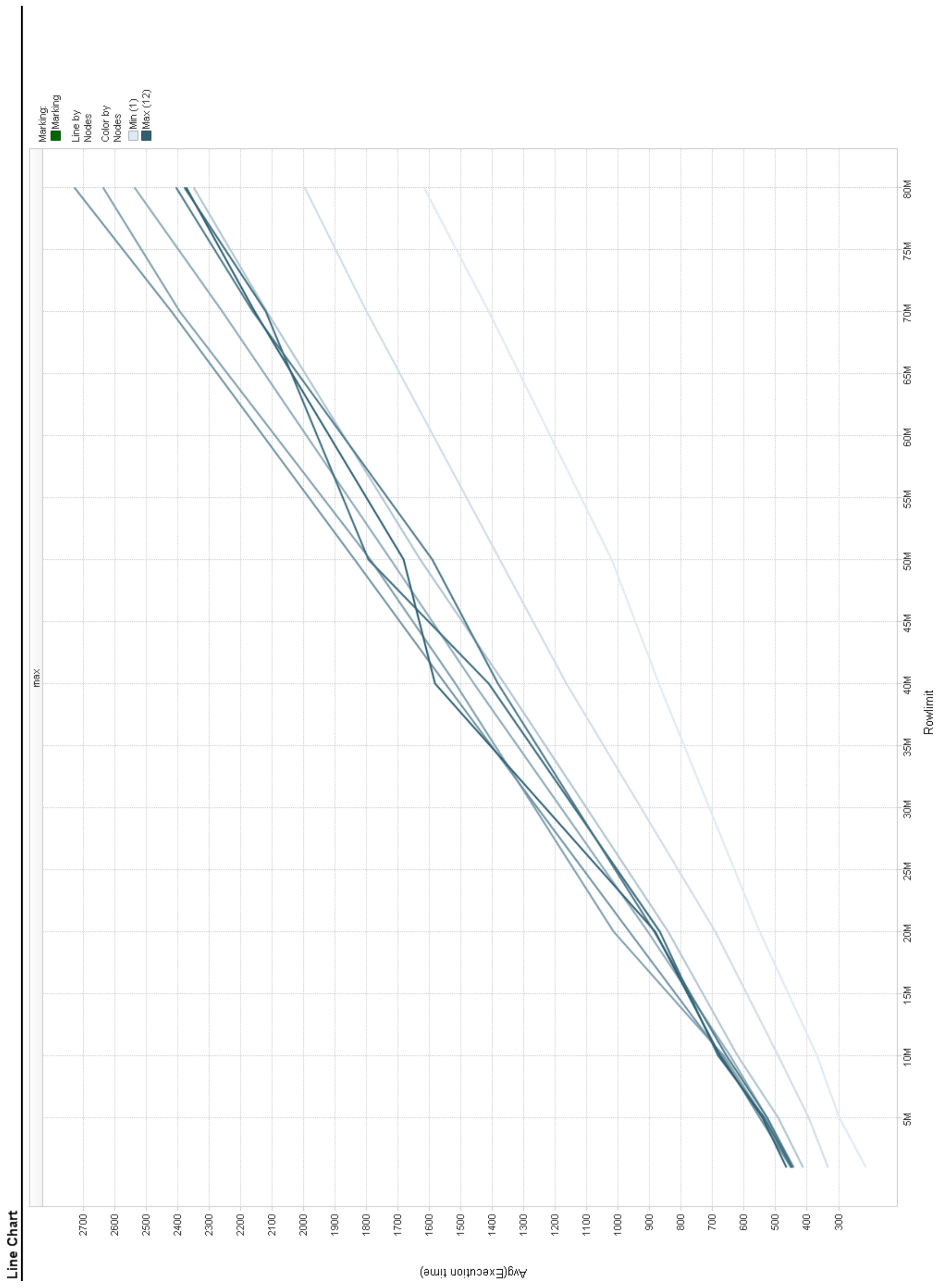

Figure 26: Average response time on column 1 per rows / node

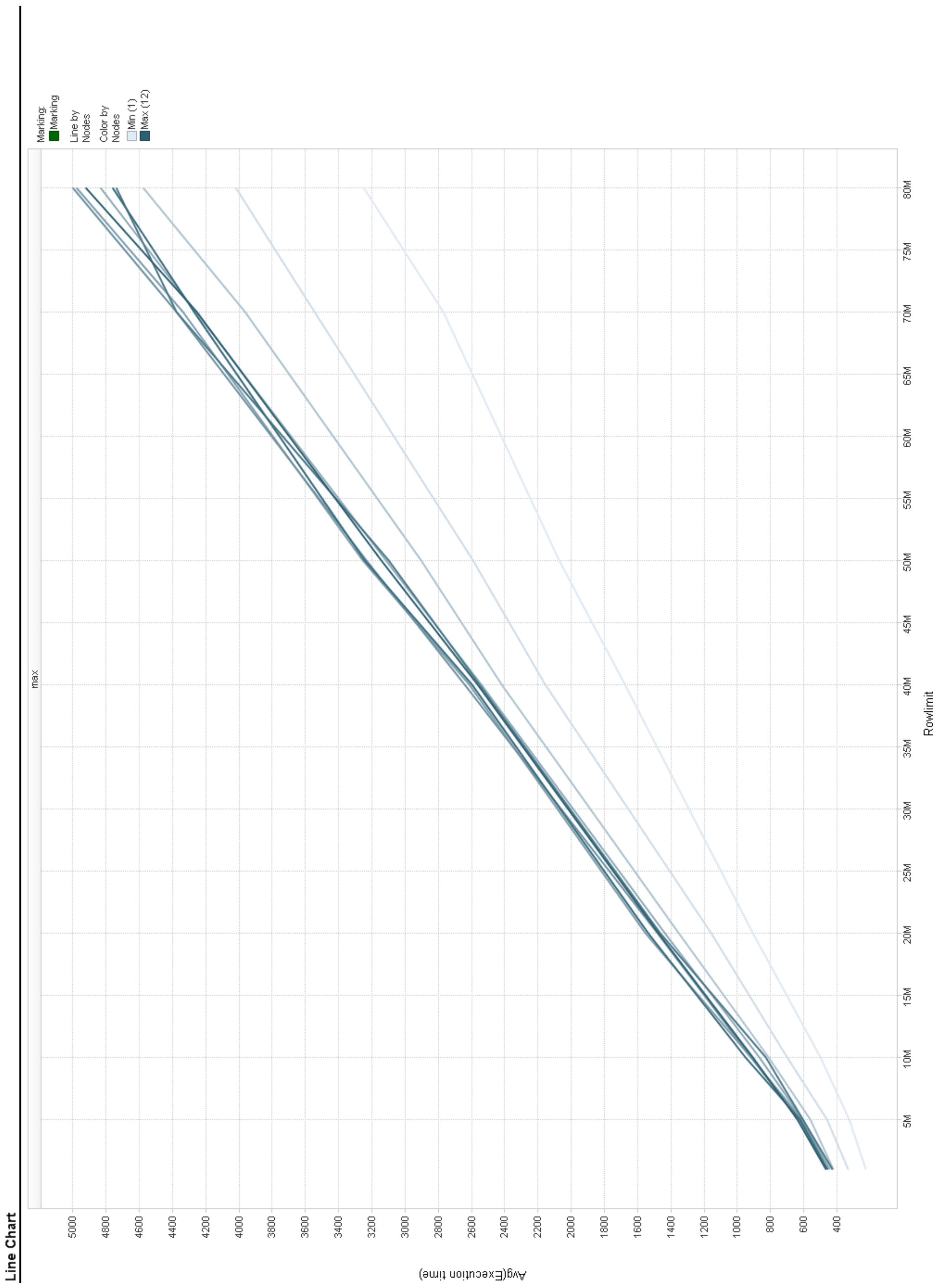

Figure 27: Average response time on column 1 per rows / node

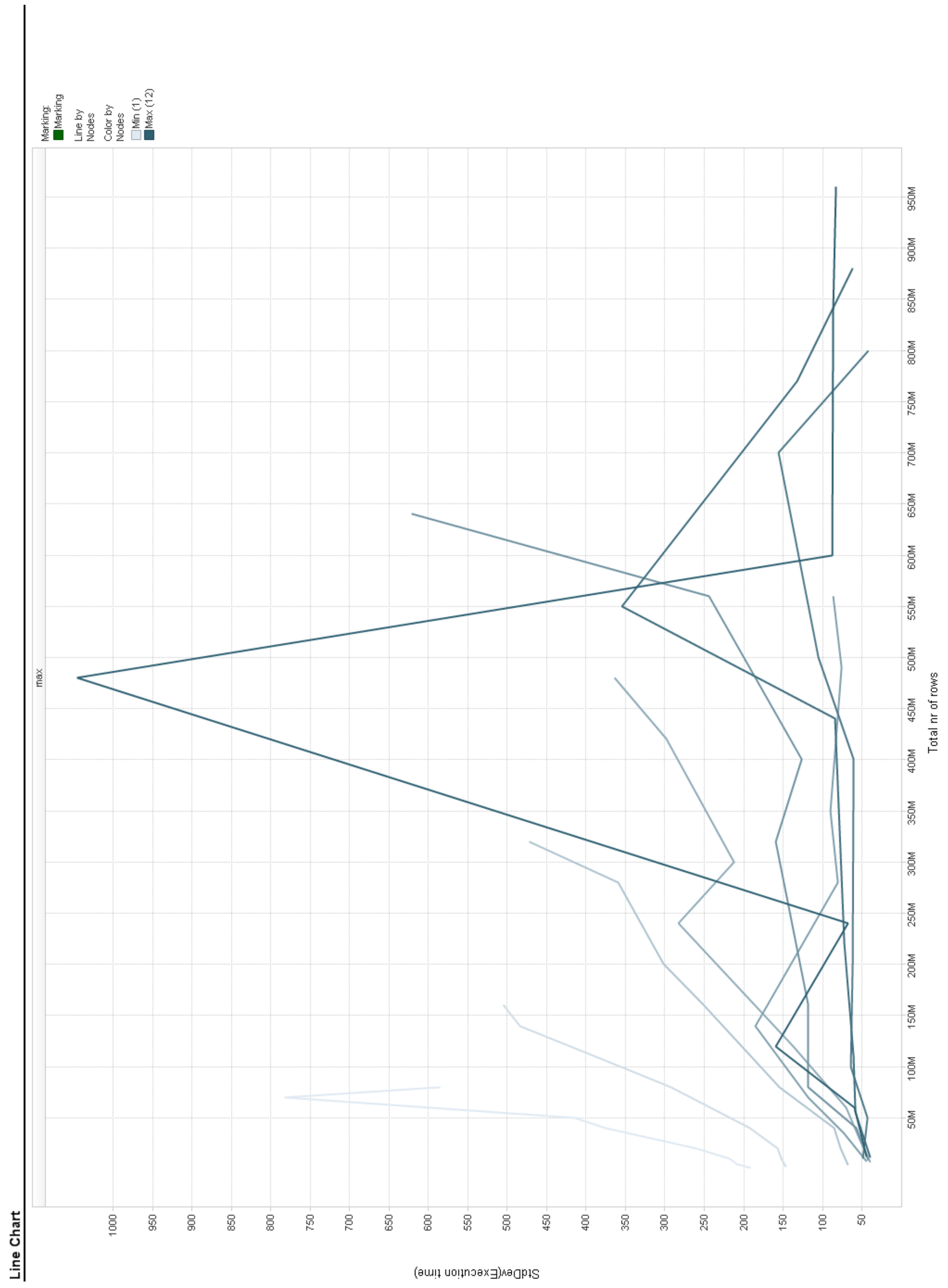

Figure 28: Standard deviation on column 1 per total nr of rows

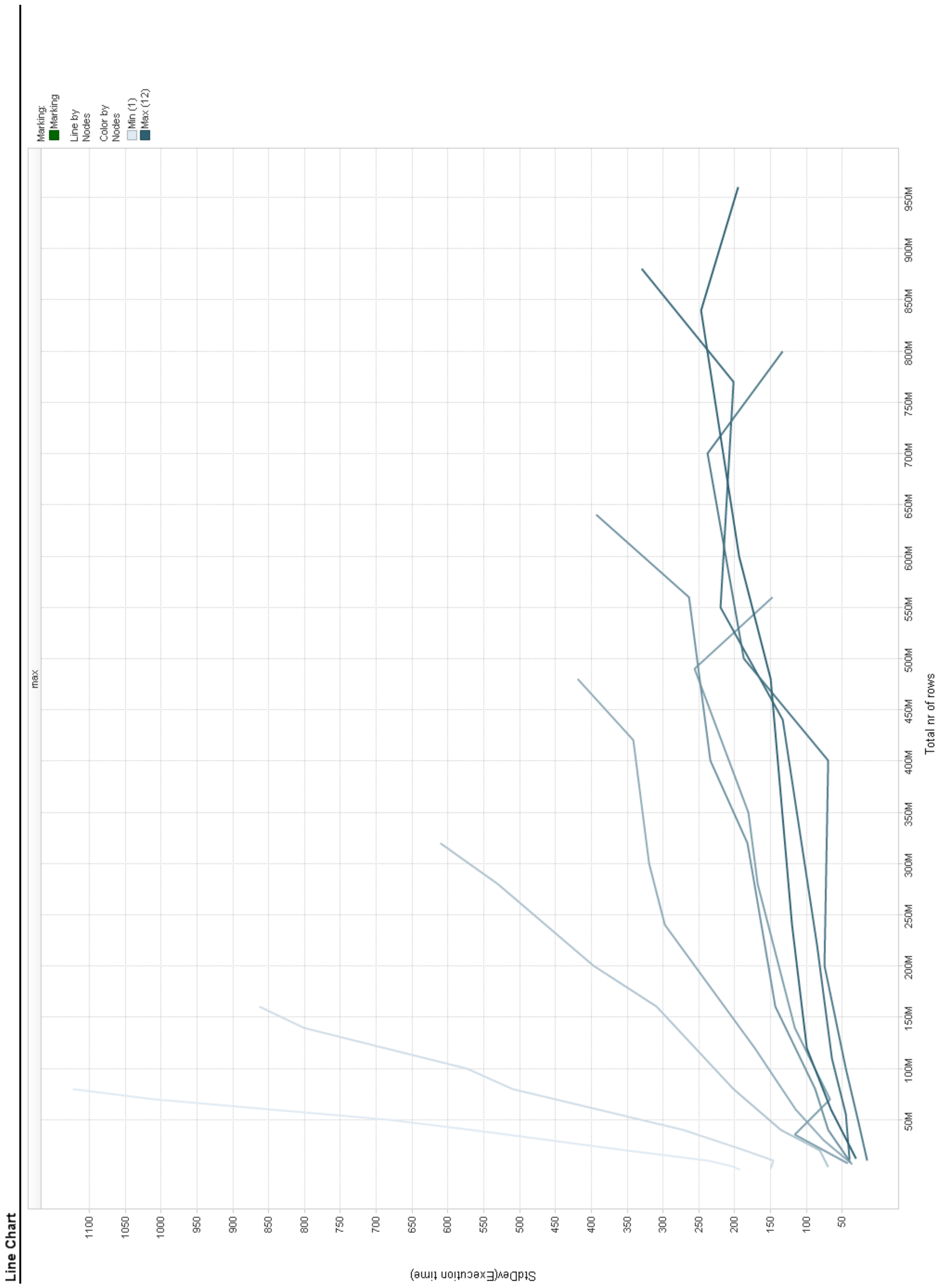

Figure 29: Standard deviaton on column 2 per total nr of rows

A.4 Median

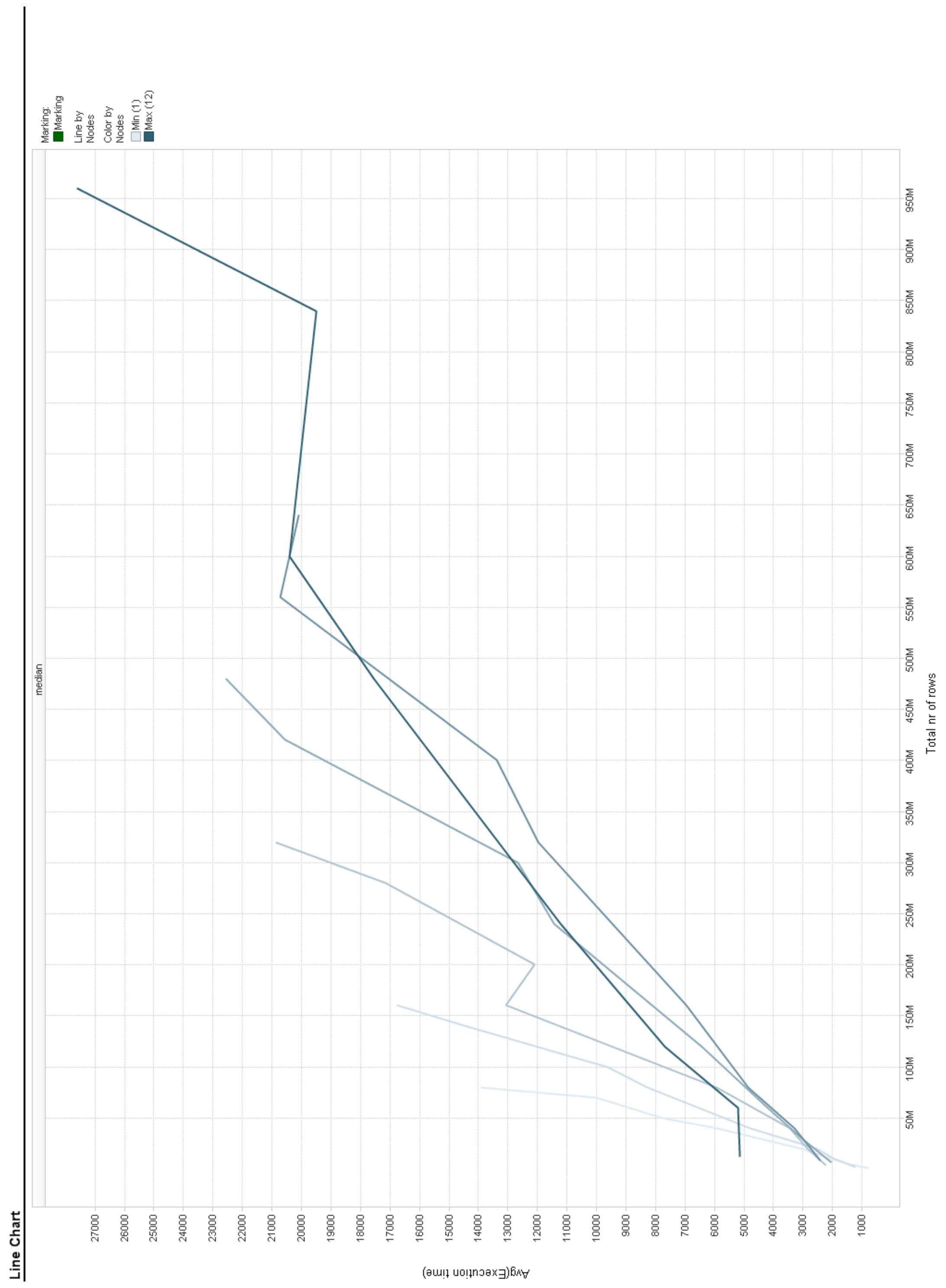

Figure 30: Average response time on column 1 per total nr of rows

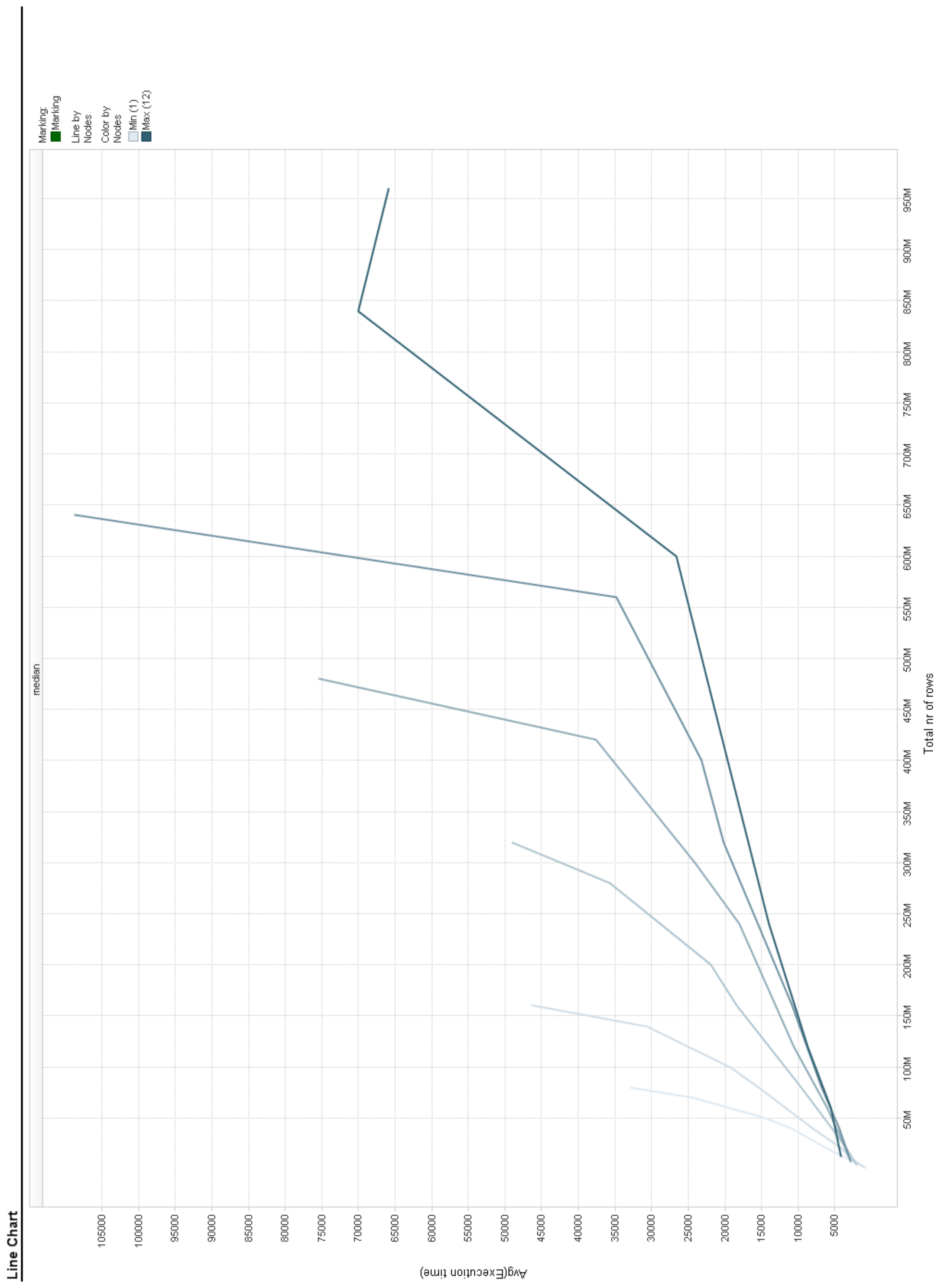

Figure 31: Average response time on column 2 per total nr of rows

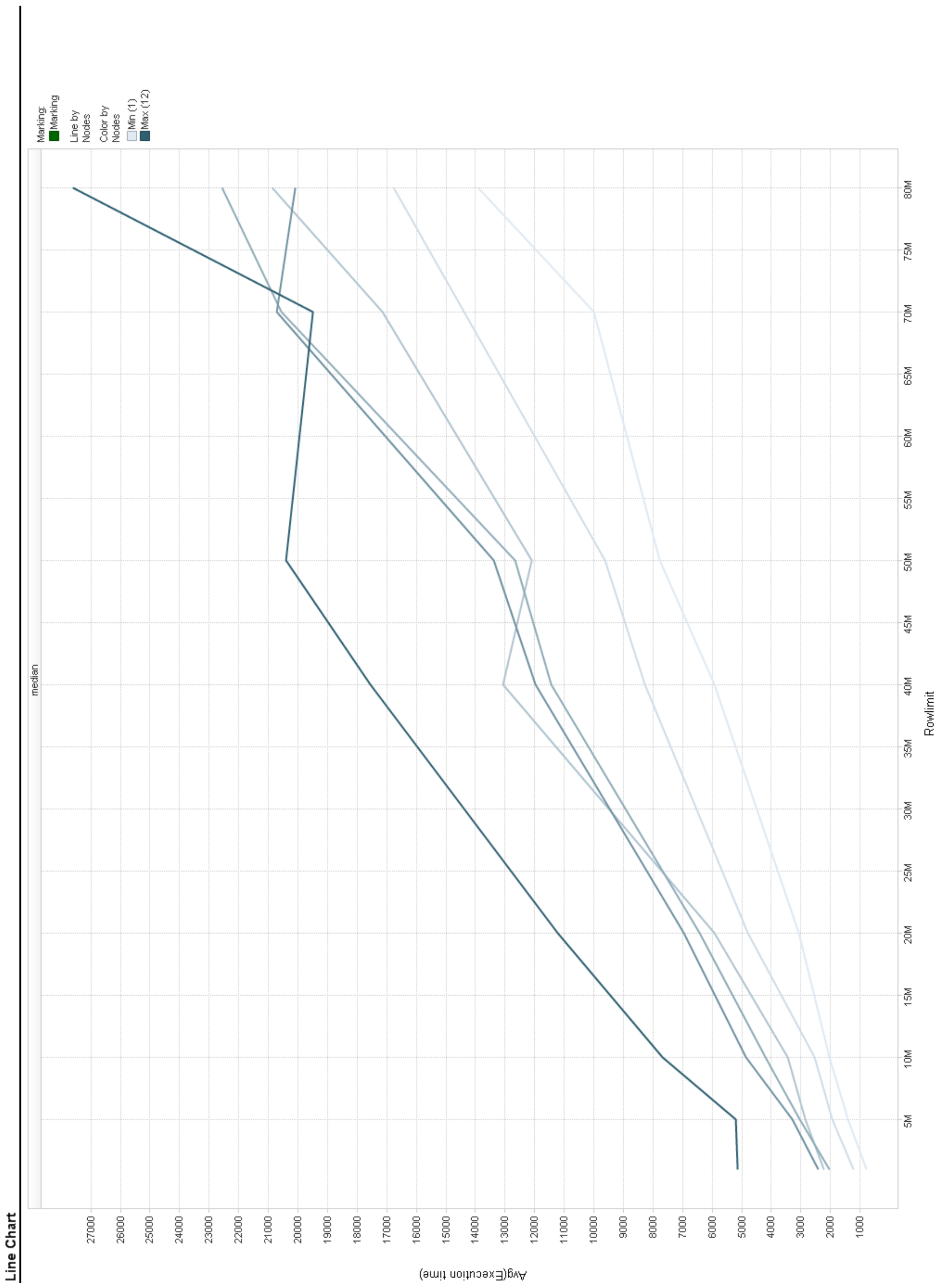

Figure 32: Average response time on column 1 per rows / node

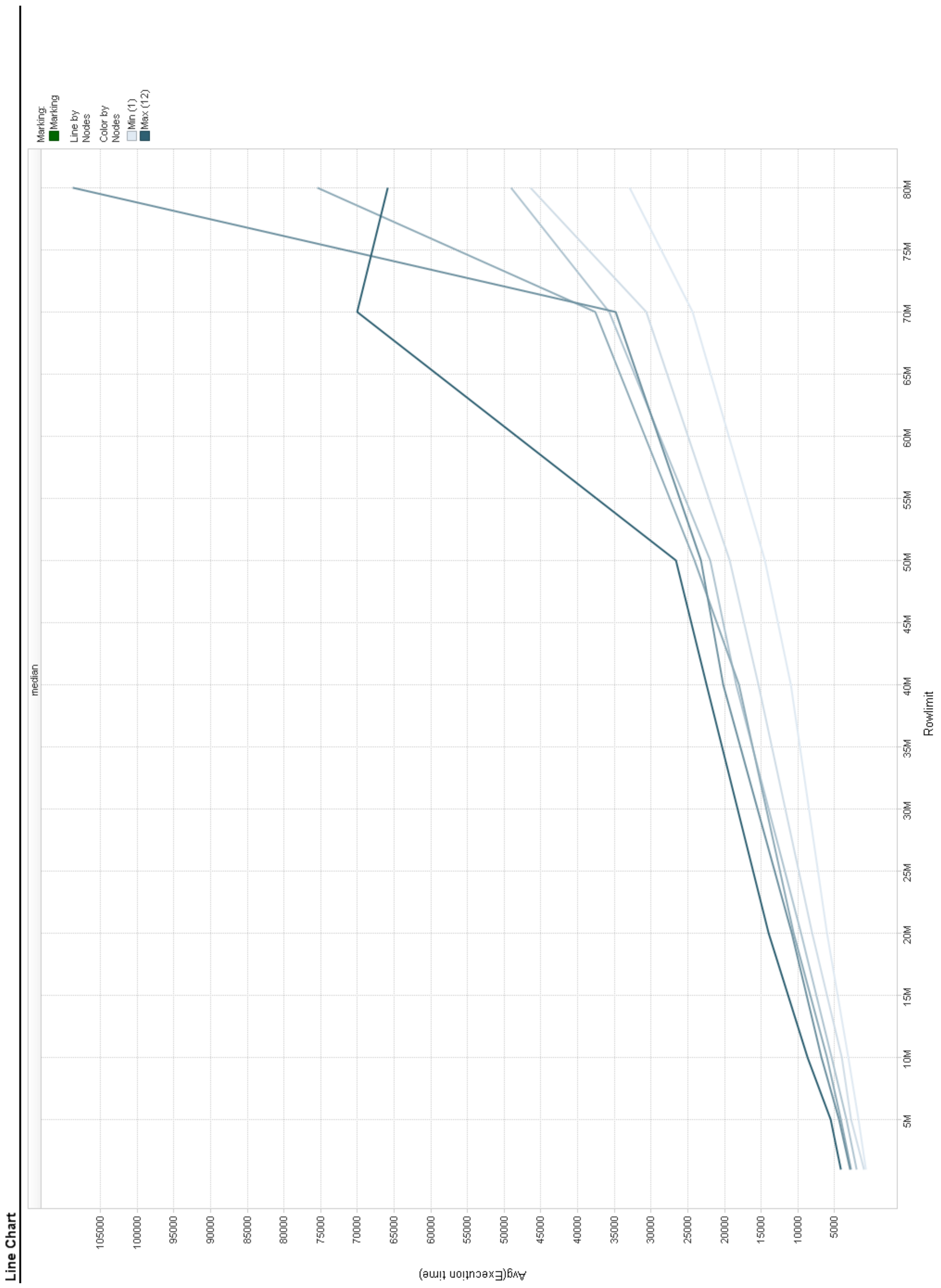

Figure 33: Average response time on column 1 per rows / node

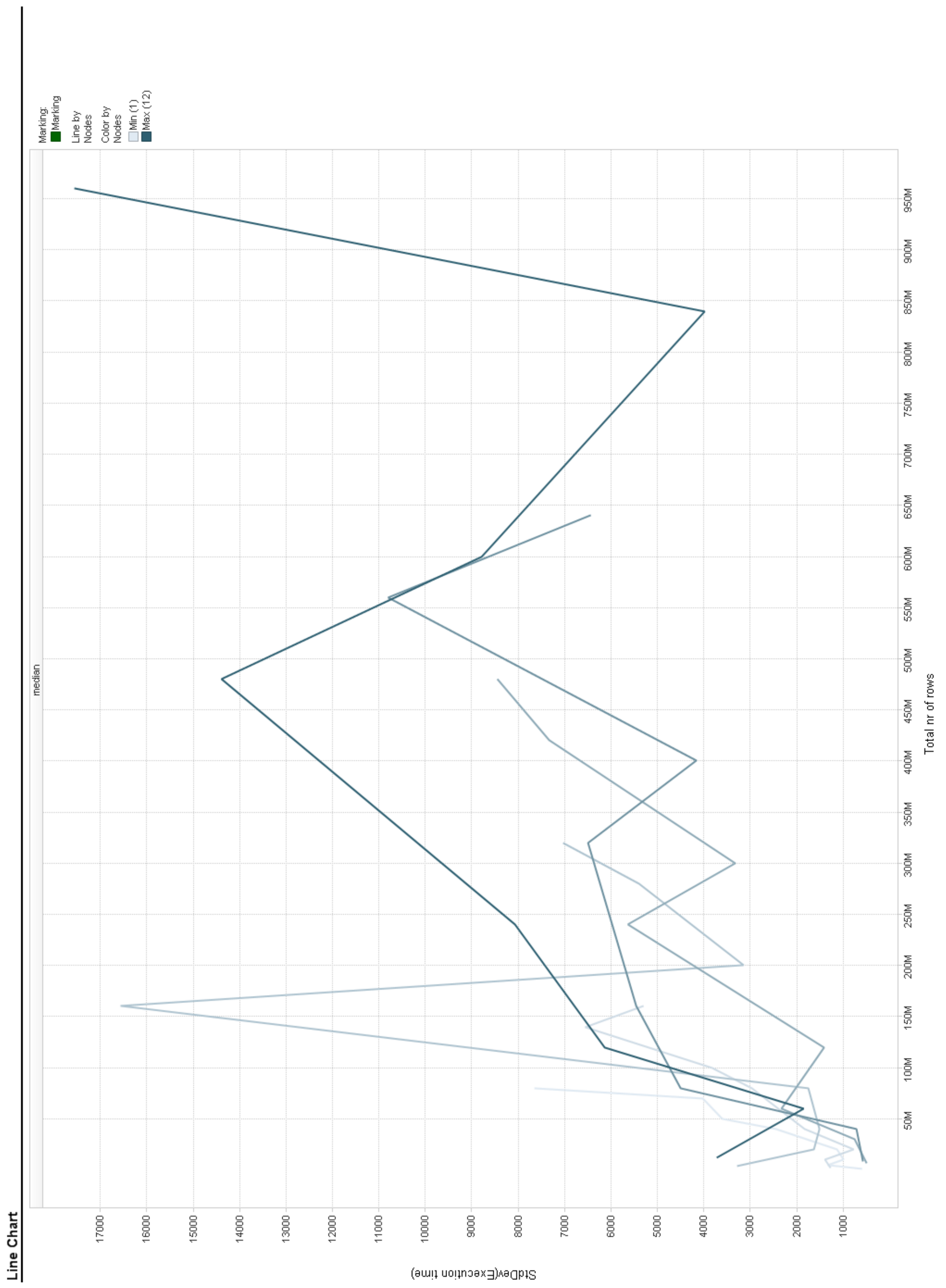

Figure 34: Standard deviation on column 1 per total nr of rows

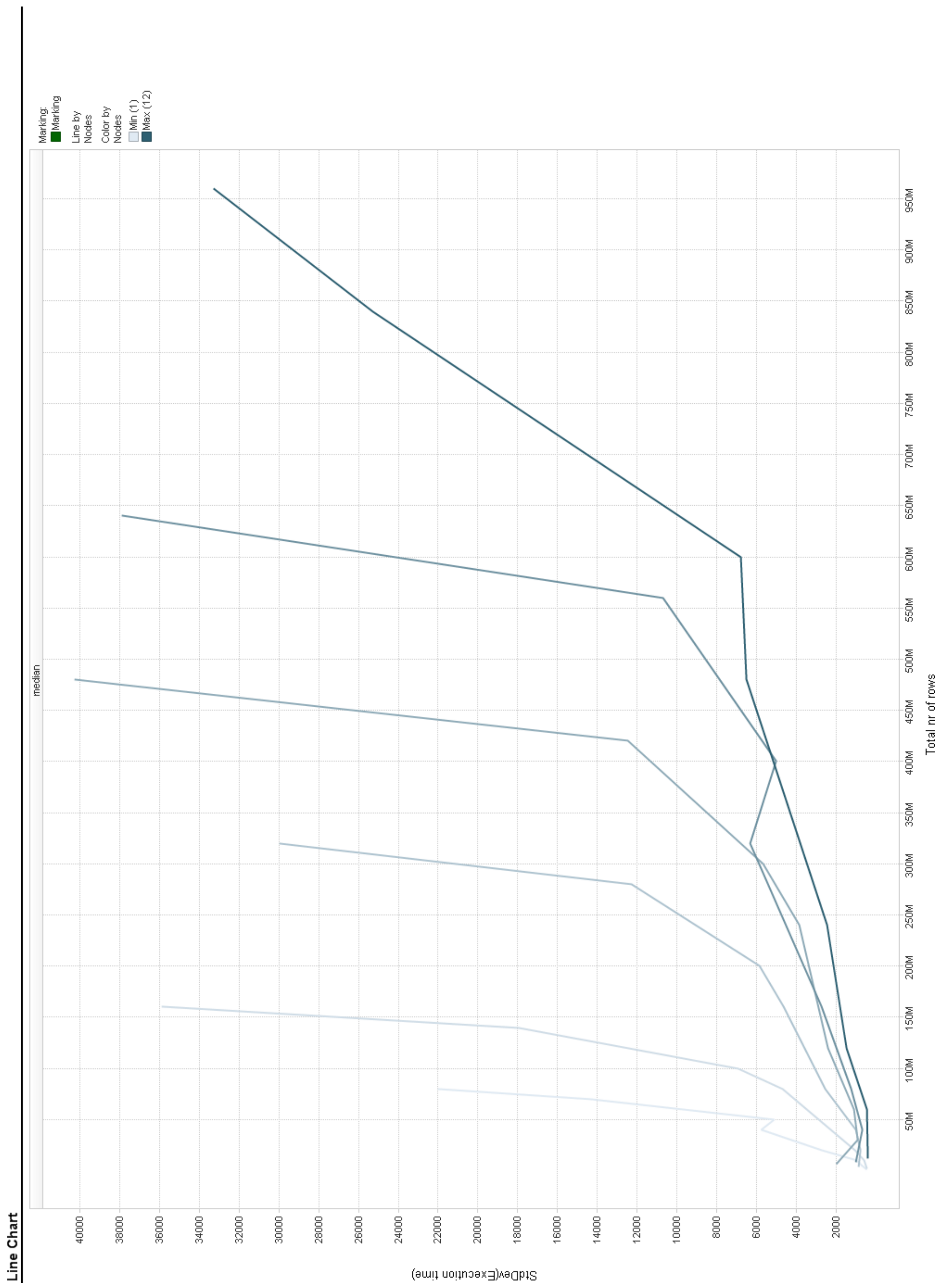

Figure 35: Standard deviaton on column 2 per total nr of rows

A.5 Sum

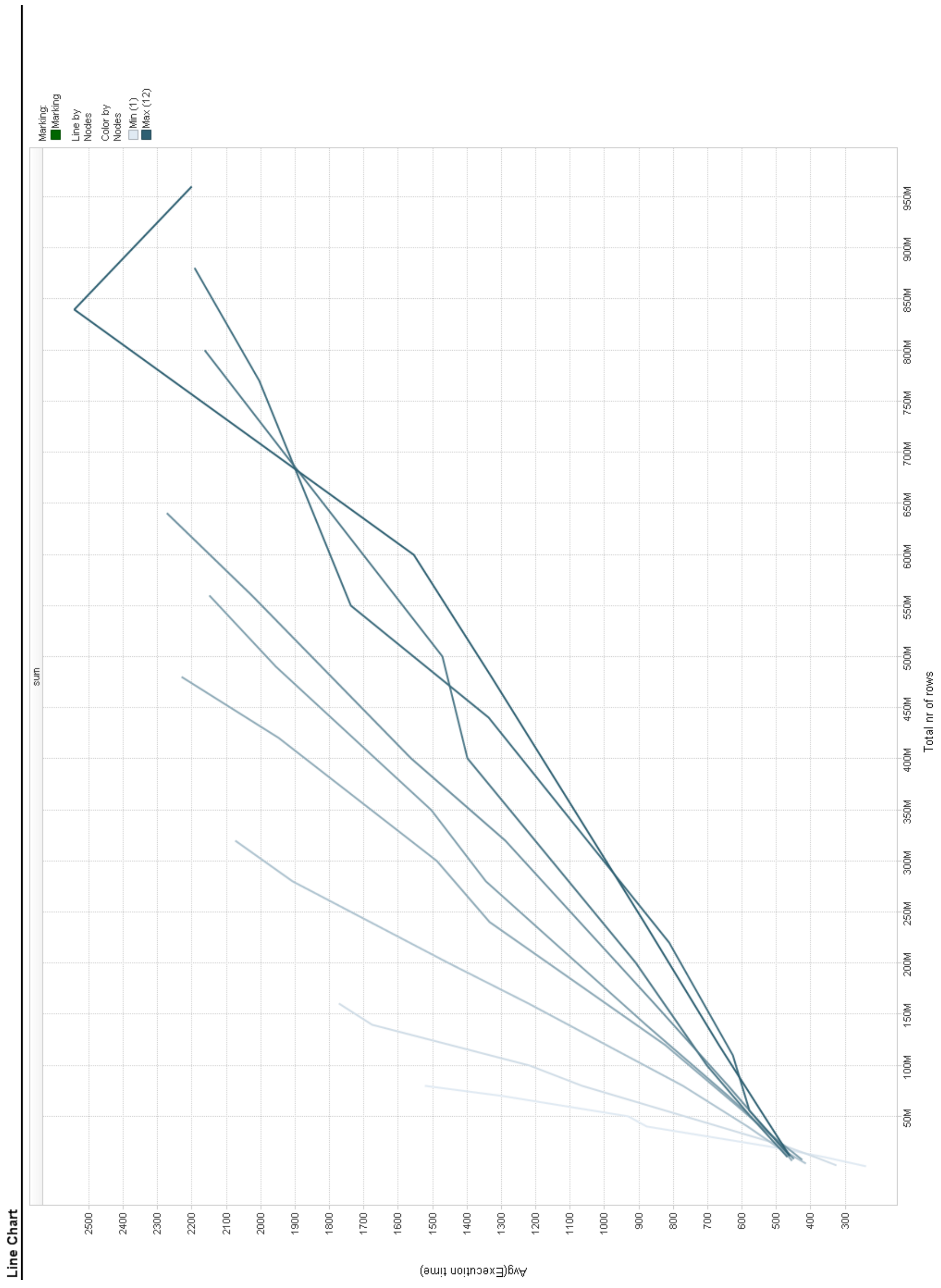

Figure 36: Average response time on column 1 per total nr of rows

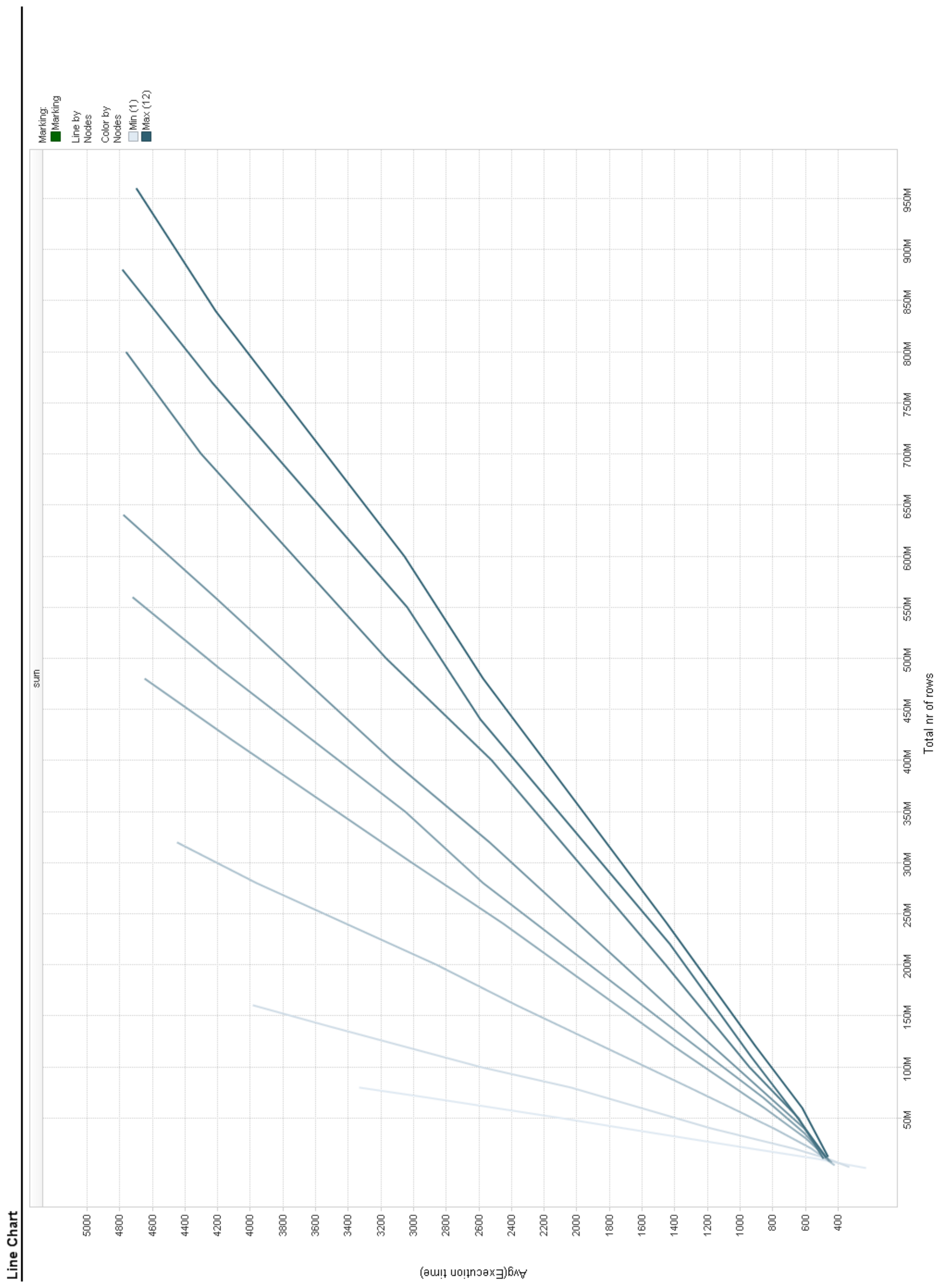

Figure 37: Average response time on column 2 per total nr of rows

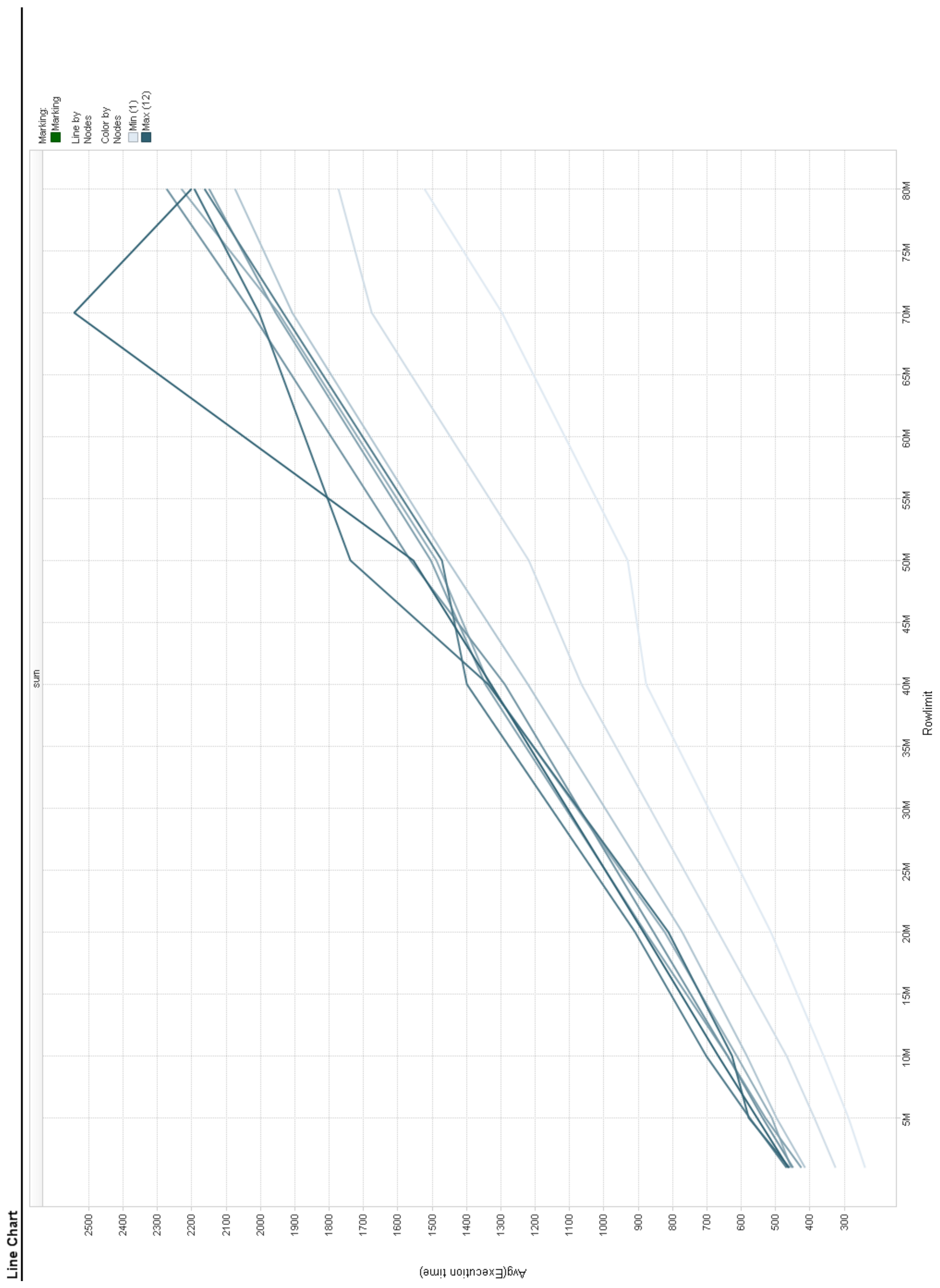

Figure 38: Average response time on column 1 per rows / node

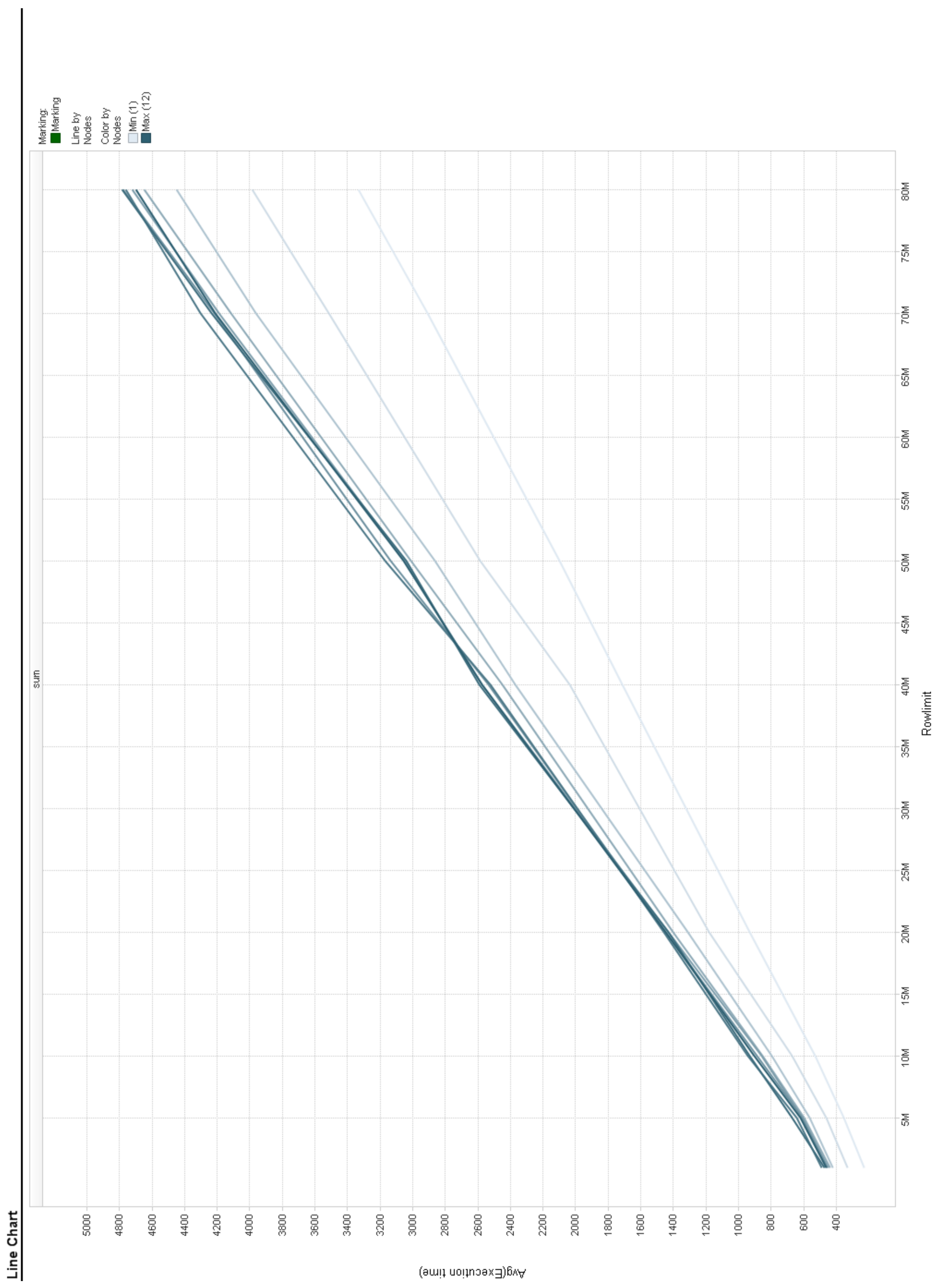

Figure 39: Average response time on column 1 per rows / node

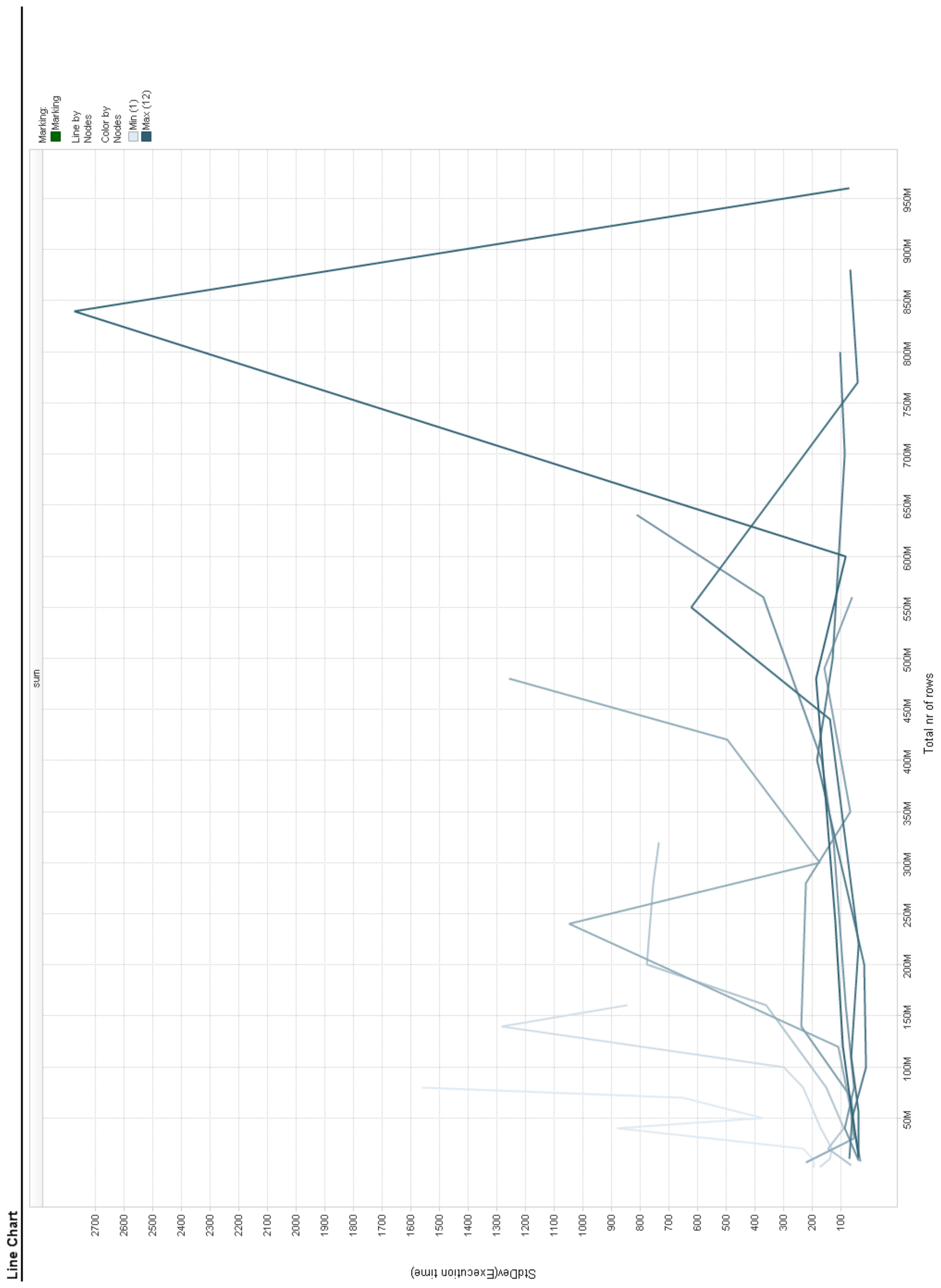

Figure 40: Standard deviation on column 1 per total nr of rows

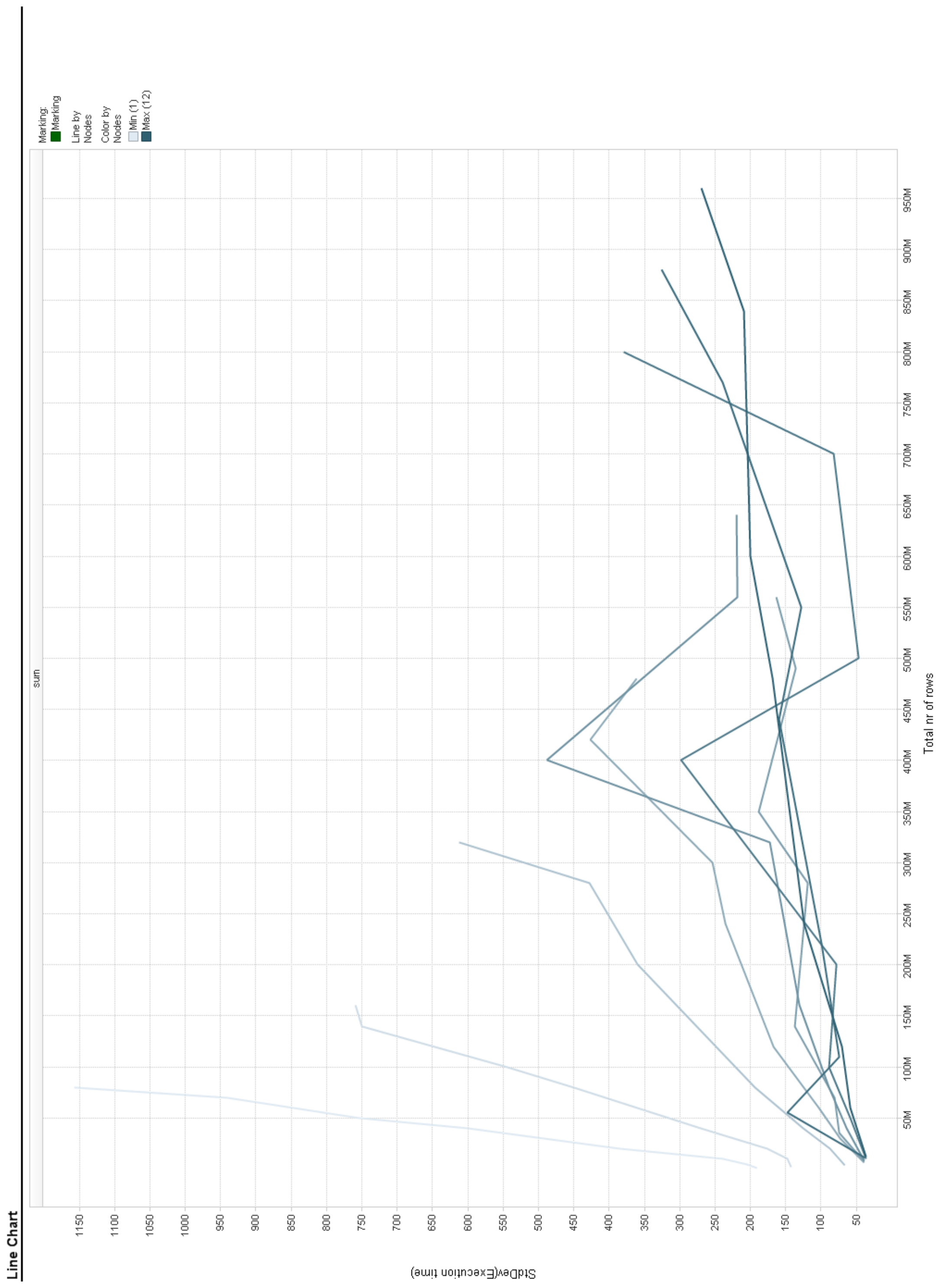

Figure 41: Standard deviaton on column 2 per total nr of rows

A.6 Unique

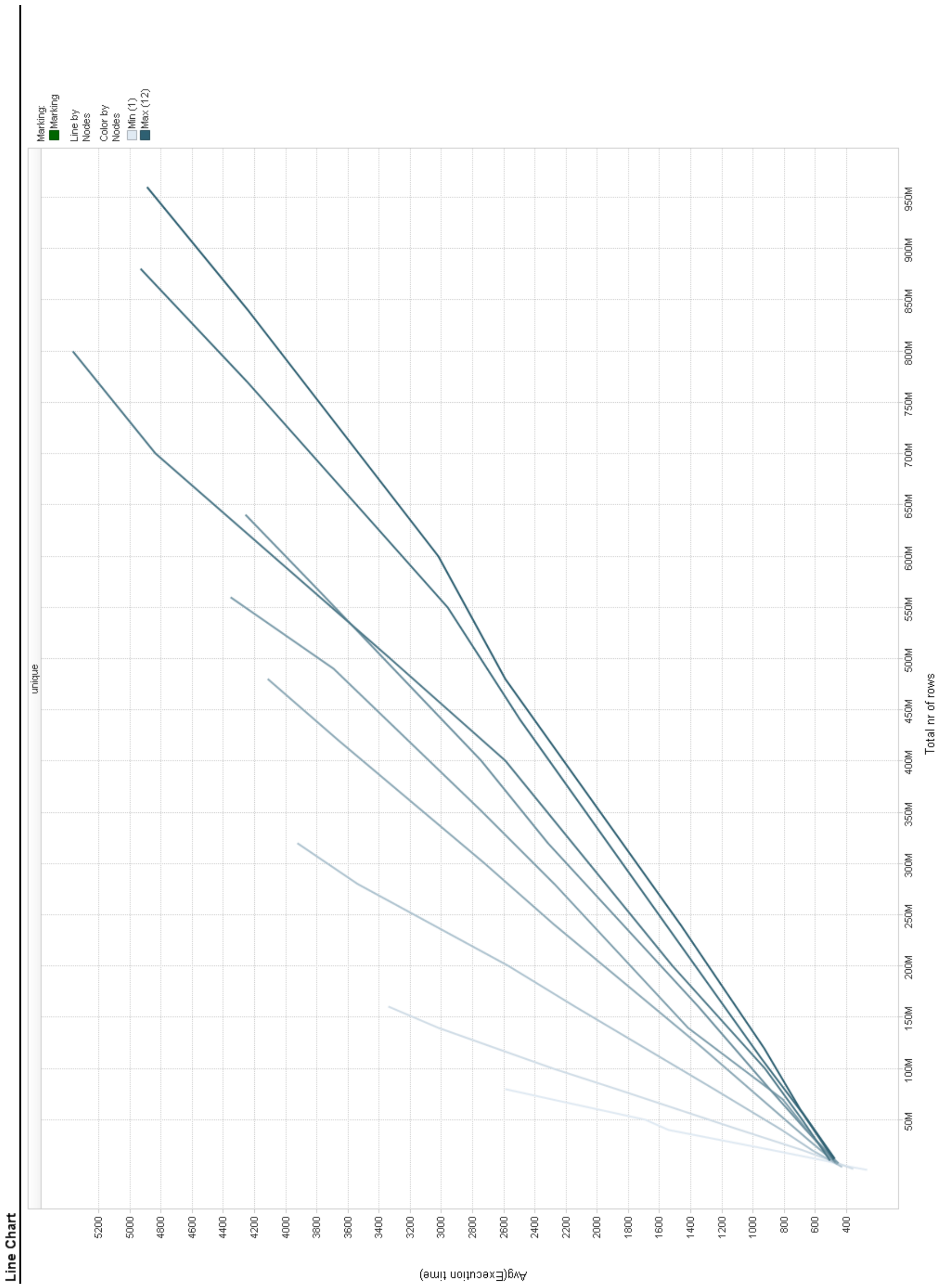

Figure 42: Average response time on column 1 per total nr of rows

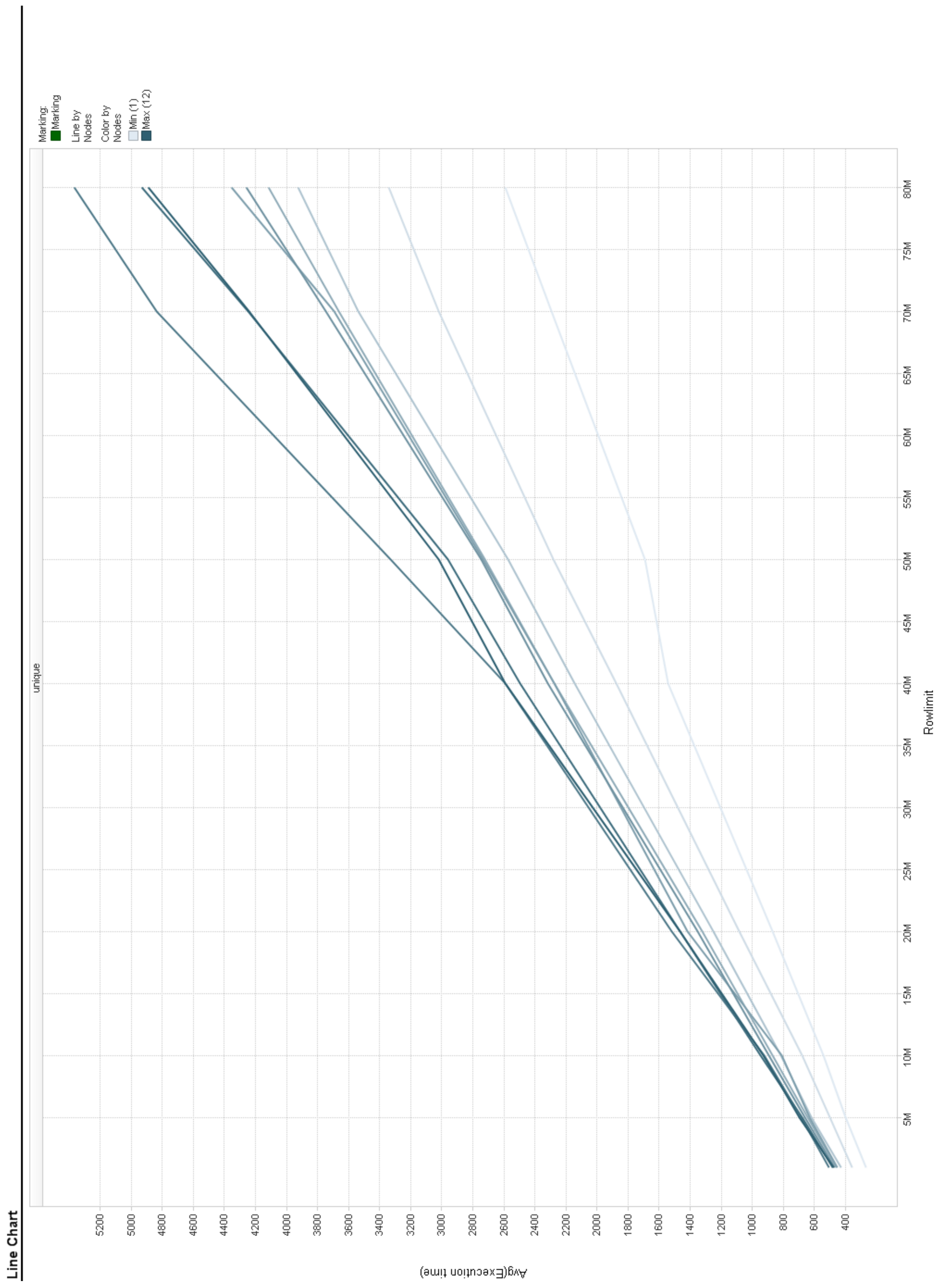

Figure 43: Average response time on column 1 per rows / node

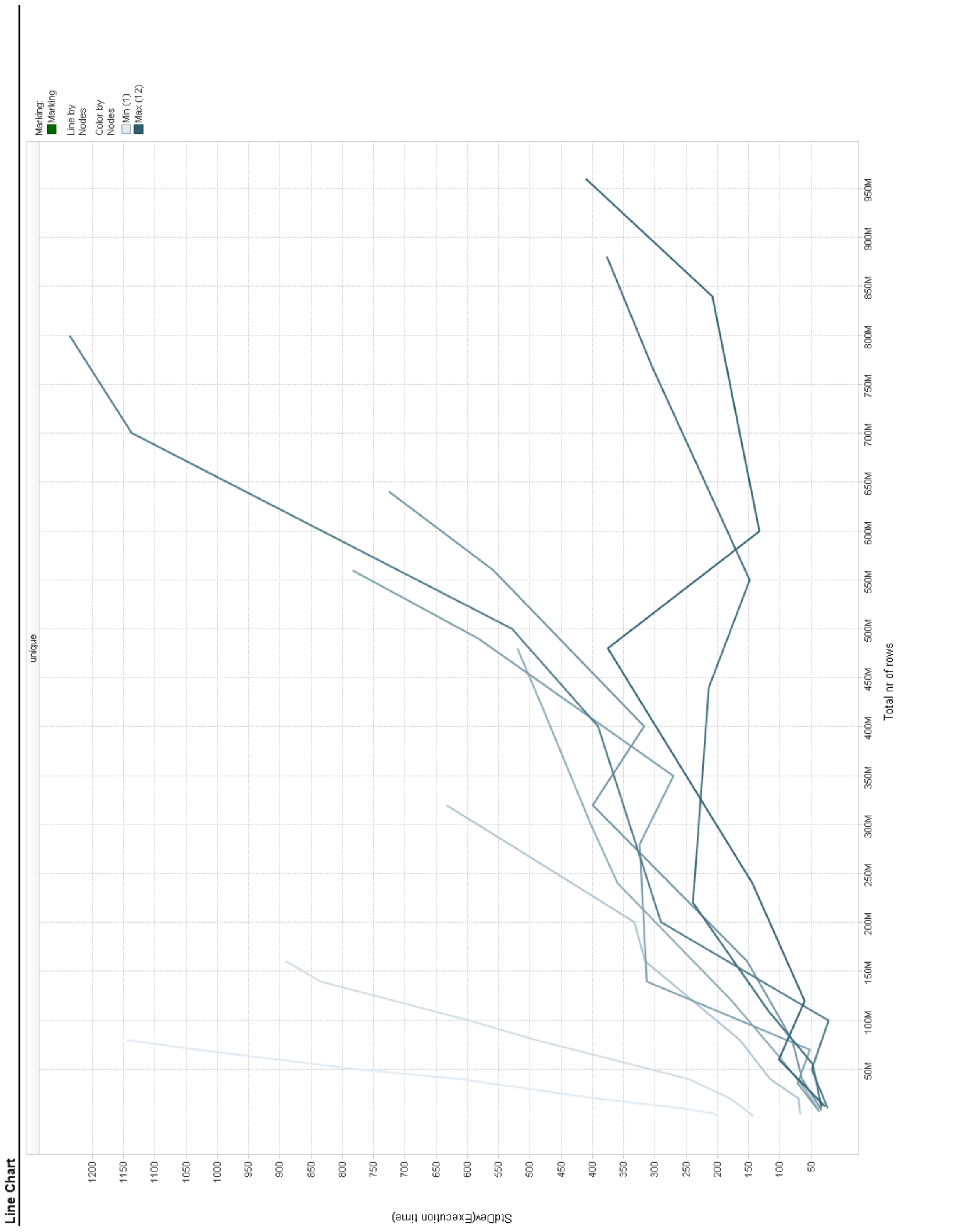

Figure 44: Standard deviaton on column 1 per total nr of rows