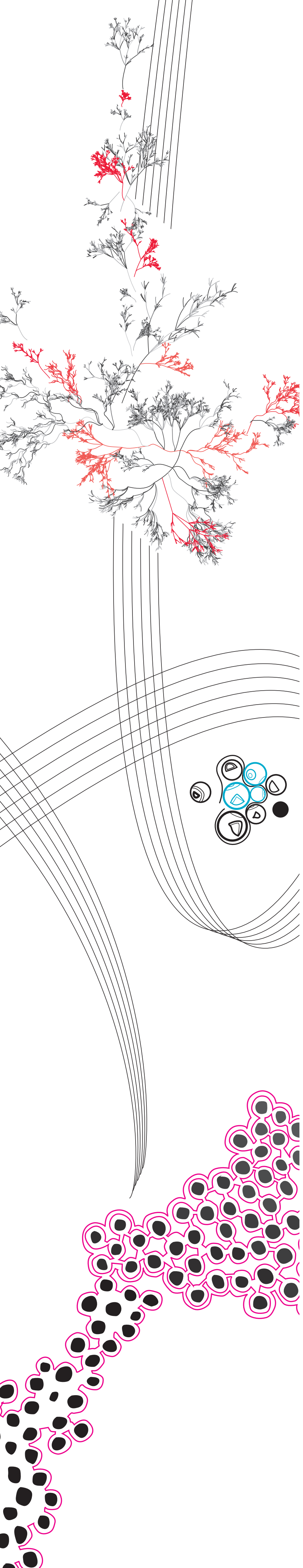


July, 2014

Master Thesis

Improving Query Performance of Holistic Aggregate Queries for Real-Time Data Exploration

Dennis Pallett
(s0167304)

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Chair Databases

Exam committee:
dr. ir. Maurice van Keulen
prof. dr. Peter M.G. Apers
dr. Andreas Wombacher
ing. Jan Flokstra

UNIVERSITY OF TWENTE.

Abstract

This thesis reports on the research done into improving query performance for holistic aggregate queries for real-time data exploration. There are three types of aggregate functions: distributive, algebraic and holistic. Distributive and algebraic aggregate functions can be computed in a distributive manner thereby making it possible to improve the performance of queries with such functions using the distributive property. However holistic functions cannot be computed distributively which is why a different method for improving the performance is needed. This research has been an attempt at this, for the ultimate goal of improving the performance of such queries to allow real-time exploration of data sets and holistic aggregate functions. Due to the complexity of such functions research has focused on a specific type, namely the n^{th} percentile functions whereby the median (i.e. the 50^{th} percentile) is primarily used.

Existing research has primarily focused on optimizing holistic aggregate queries by computing approximate results with a certain error bound or guarantee on the chance of an exact result. Most existing algorithms use sampling or (wavelet) transformations to improve query performance but these techniques will never be able to return exact results in every case. In this research focus has been on finding methods for improving query performance whilst always returning exact correct results.

Results of this research has shown that the biggest factor that influences the performance of a database query are disk operations, whereby two different types of disk operation can be differentiated: seeking and scanning. A scanning operation is primarily limited by the bandwidth of the disk, i.e. how many bytes can be read per second, whereas the seeking operation is limited by how fast the physical head of the disk can be moved to the correct location. Any database query that requires fetching a lot of data from disk will have its performance severely limited by the bandwidth of the disk. In most cases disk seeking is very limited, provided the correct indexes are setup. Operations done in-memory, such as sorting a very large dataset, can also have a negative effect on the performance of a query. Although generally operations in-memory are considered very fast, they cannot be ignored when dealing with query performance.

Investigation into the current performance of median queries has shown that the basic algorithm for processing such queries, currently used by most database engines, has two main problems which are the cause of bad query performance and poor scalability: (i) *all* the tuples needed to calculate the median of a query must be fetched from disk and (ii) all these tuples must be stored and sorted in-memory before the exact median can be returned. For very large datasets and large queries, involving hundreds of thousands of individual tuples, this algorithm has very poor performance.

To solve these problems an alternative algorithm has been developed called the *shifting algorithm*. This algorithm relies on pre-computed binary data streams and specific ”shift factors” to avoid the need to fetch all tuples from disk or to do any in-memory storage or sorting. Experimental test results have shown that this algorithm is able to improve the query performance of median queries by several orders and improves the overall scalability of such queries.

Unfortunately the shifting algorithm is unsuitable for data sets with multiple dimensions, as is often the case in the field of OLAP and data warehousing, due to the increased complexity of multiple dimensions. Therefore another alternative algorithm called the *full scanning algorithm* has been developed. This algorithm relies on a pre-computed pre-sorted binary stream of the full data set but does not require the use of ”shift factors”. This algorithm does need to fetch all tuples from disk and store them in-memory but does so in a more efficient way than the basic algorithm due to the pre-computed binary data stream. Additional, this algorithm does not need to sort in-memory all the tuples to determine the median. This algorithm is able to significantly improve the query performance of median queries but does not improve the scalability of such queries.

To also improve the scalability a variant of the full scanning algorithm has been developed which depends on calculating the number of tuples affected by a query before calculating the median. With this extra piece of information (i.e. the tuple count) the full scanning algorithm is able to reduce the scanning time and avoids the need to store all tuples in-memory. Test results have shown that this variant is able to improve absolute query performance even further and also improves the scalability of median queries in a similar way as the shifting algorithm. It is important to note however that this variant depends on the availability of an efficient count operation.

Two additional optimizations have been developed which can be used together with all of the alternative algorithms to further improve query performance. The first optimization, called the *with-values optimization*, relies on including the actual measure values into the pre-computed data structures used by the algorithms. This avoids the need to fetch the actual values, once the correct tuple has been identified as the median, and allows the algorithms to process median queries without the need for the original dataset. This is especially beneficial in an OLAP context, as it means that very large datasets can remain in the data warehouse whilst the end-user can still explore this dataset using the much smaller pre-computed data structure. Test results have shown that query performance can be improved with this optimization but can also be deteriorated.

The second optimization, called the *sub-groups optimization*, is focused on improving the performance of queries that cover a (very) small part of the full data set. Test results have shown that the performance of such queries may actually worsen when one of the alternative algorithms are used instead of the basic algorithm. To solve this problem the sub-groups optimization relies on pre-computed additional data structures which are specifically tailored towards smaller queries thereby improving their performance. Test results have shown however that this effect is quite limited and requires a lot of storage. Therefore a very careful consideration must be made before deciding to use this optimization.

Finally a framework, the *algorithm selection framework*, has been developed to aid with the selection of the right algorithm and optimizations. This framework is used to provide a clear guidance in which case each algorithm and optimization should be used. It consists of an easy-to-use decision tree and has a solid empirical background because it is based the experimental test results and evaluation of the algorithms and optimizations. Using the framework the right algorithm and optimizations are selected for every circumstance and/or data set.

This research has conclusively shown how the query performance and scalability of median aggregation queries can be improved. Investigation has shown that there is no single algorithm that is suitable for this task but instead a set of different algorithms, optimizations and a selection framework has been developed that are able to improve query performance for a varying range of use-cases.

Table of Contents

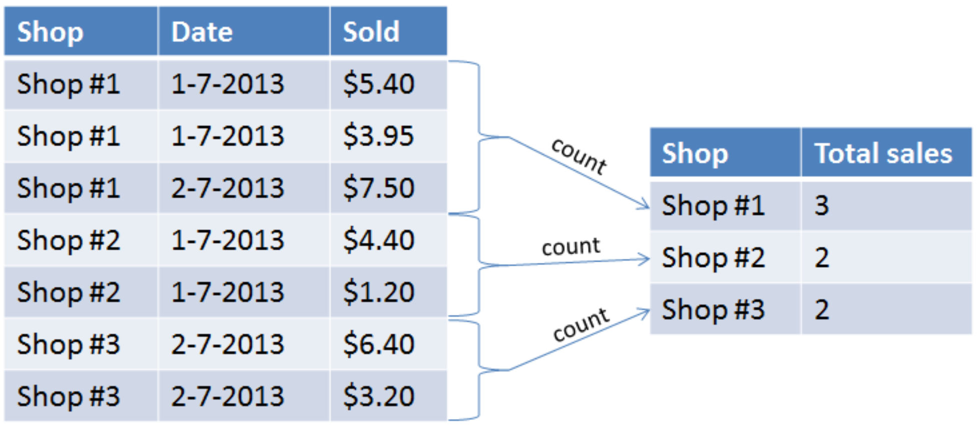
List of Figures

List of Tables

Chapter 1

Introduction

Analysis and exploration of data stored in a database management system often involves the use of aggregation operations. A simple example of an aggregation operation, shown in figure 1.1, would be to retrieve a list of total number of sales per shop (for a database that contains all the sales for a chain of shops) whereby an aggregation function is used to count the number of sales for each unique shop. An aggregation function is a function that is used to summarize a set of data tuples into a single value and is often used in conjunction with a grouping operator to execute the aggregate function for a specific set of groups, exactly as in the shops example. The most simple aggregation function is the count function which counts all the tuples in the data set and returns a single integer value: the total count of all the tuples or, if used in conjunction with a grouping operator, a count of tuples for each group.



Shop	Date	Sold
Shop #1	1-7-2013	$5.40
Shop #1	1-7-2013	$3.95
Shop #1	2-7-2013	$7.50
Shop #2	1-7-2013	$4.40
Shop #2	1-7-2013	$1.20
Shop #3	2-7-2013	$6.40
Shop #3	2-7-2013	$3.20

Shop	Total sales
Shop #1	3
Shop #2	2
Shop #3	2

Figure 1.1: A simple aggegration example

Aggregation functions can be divided into three distinct types [16]:

Distributive An aggregate function is distributive when it can be computed in a distributed manner. For example, assume that a data set is partitioned into *n* sets. The distributive aggregate function is applied to each set resulting in *n* aggregate values. If the aggregate value of the whole data set can now be computed by applying the same aggregate function to all the set aggregate values then the aggregate function is distributive. Examples of distributive aggregate functions are sum(), count(), min(), and max().

Algebraic An aggregate function is algebraic if it can be computed by an algebraic function with M arguments (where M is a bounded positive integer) whereby each argument can be obtained with a distributive aggregate function. An example of an algebraic aggregate function is the average() function, which can be computed with the following algebraic expression: $\frac{sum()}{count()}$, with both sum() and count() being distributive.

Holistic An aggregate function is holistic if it cannot be described with an algebraic function with M arguments. An example of a holistic aggregate function is the median() function, which cannot be described by an algebraic expression or calculated in a distributive manner.

Aggregation operations are most often used for data warehousing, data mining and OLAP purposes [7]. One important aspect of these purposes is that they often concern large amounts of data, easily reaching millions or billions of individual data tuples stored in the database [27]. Computing the results for a database query with an aggregation function and possibly a grouping operator for these kinds of data sets can take a long time, ranging from a few seconds to possibly minutes or hours depending on the size of the data set and the complexity of the query [2]. Therefore if any (near) real-time analysis is required the performance of these aggregation queries must be improved by several orders to make this possible [19].

Many existing approaches for improving performance depend on pre-calculated structures (such as indexes) or higher-level views of the full data set (i.e. pre-aggregated structures). However these techniques are only suitable for distributive and algebraic aggregate functions because they rely on the distributive property of the aggregation function, meaning that the result of the function can be calculated using (pre-aggregated) sub-parts instead of all the individual elements thereby avoiding the need to access all individual elements. This is not the case for holistic functions because the result of any holistic function cannot be computed using pre-aggregated sub-parts and instead generally requires access to all the individual elements. Many existing techniques for improving query performance for these types of aggregation functions use approximation whereby an exact result is never computed but an approximation with a certain error bound or guarantee. These techniques are most often based on the usage of sampling and can be used for holistic aggregation functions but will never be able to provide an exact result.

The goal of this research is to develop a new algorithm that will improve the performance of queries that contain a holistic aggregate function without sacrificing accuracy of the results. Due to the complexity of holistic functions it is assumed that a single algorithm that improves the performance for each holistic aggregation function does not exist or delivers very little performance improvement. Therefore this research focuses on a specific type of holistic function, namely finding the kth smallest element in a dataset.The most well-known example for this type of function is the median, which always returns the $\frac{N}{2}$th smallest element for a dataset with N elements, i.e. the middle element or an average of the 2 middle elements when there are is even number of elements in the dataset. Other common examples are quintiles (1st, 2nd, 3rd and 4th) and percentiles.

The direction of this research is primarily focused on reducing the need for the aggregation function to access all the individual tuples, as is usually the case for holistic aggregation functions. Although holistic functions cannot be calculated in a distributive way, and can therefore not rely on pre-aggregated sub-parts, other (existing) techniques are likely to be available and can be combined to reduce or eliminate the need to access all individual tuples.

1.1 Motivation

This research is part of the COMMIT project, which is a public-private research community aimed at solving many great challenges in information and communication science [9].More specifically, this research contributes a small part to Work Package 5 (WP5) [10] of the TimeTrails sub-project of the COMMIT project. This work package is focused on designing and developing query processing techniques to support real-time trajectory exploration and data mining. Currently query processing techniques have been developed and evaluated to support real-time exploration of data when a distributive or algebraic aggregation is used. However the currently developed techniques cannot be used to support real-time exploration of data when a holistic aggregation is used. The results of this thesis are aimed at making this possible.

A concrete example whereby this work could be used is for exploring a dataset of railway incidents overlaid over a country map. Currently it is already to possible to show the average number of incidents per area but averages are very susceptible to outliers therefore it would be very useful to also include the median number of incidents (or the 25th and 75th percentile to get a lower- and upper-bound) per area to provide the end-user better insight into the data set.

1.2 Research Question

The main research question is as follows:

> *How can the performance of queries with the median aggregate function be improved whilst still returning accurate results?*

To be able to answer the main research question the following sub-questions will be considered as part of this research:

What factors have an influence on the performance of a database query?
Before the performance of median aggregate queries can be improved it is absolutely necessary to have a thorough understanding of the most important factors that have an influence on the performance of database queries and how they can be measured as accurately as possible.

Which algorithms are able to deliver improved query performance?
The sub-question will lead to a clear description of several new algorithms that are able to compute the exact results of a median query whilst improving the query performance.

Which algorithms are most suitable in different circumstances?
In this research several different data sets and test queries, each with their own characteristics, will be used to simulate real-world conditions. It is very likely that these different data properties will lead to several algorithms providing the best query performance in different circumstances, therefore it is important to have a framework that is able to clearly define which algorithm is best suited for specific situations.

1.3 Structure of this document

In chapter 2 existing research related to this research and existing approaches currently used by commercial and open-source RDBMS packages will be discussed. This chapter will provide a conclusion to the first sub-question. Then, in chapter 3 the concept of query performance will be discussed. In specific, which factors influence query performance, how query performance can be measured and an empirical verification of these factors and measures. Chapter 4 outlines the exact test setup used to run experiments with regards to the datasets, test queries, hardware, RDBMS products, etc.

In chapter 5 the alternative algorithms and optimizations that have been developed as part of the research are discussed and their advantages and disadvantages. This chapter provides a conclusion to the second sub-question. In chapter 6 the alternative algorithms and optimizations are systematically evaluated on two main points: query performance and storage requirements. This evaluation is based on experimental test results and analysis. Based on the work discussed in chapter 5 and the results of the evaluation as outlined in chapter 6, the algorithm selection framework is presented in chapter 7. This chapter provides the conclusion to the third sub-question.

Finally, in chapter 8 this thesis is concluded by answering the research questions, discussing potential threats to the validity of this research and outlining possible future research directions.

Chapter 2

Related Work

This chapter provides an overview of existing work that is related to database management systems and efficient query processing with (holistic) aggregate functions. The first section contains short discussion on the different fields of research that are related and how they differ. In the second section an overview is provided of some of the most common techniques and algorithms. The final section of this chapter will look into existing RDBMS products and their existing access methods for holistic aggregate queries, if any.

2.1 Research fields

Current research into aggregation functions for finding the *k*th smallest element has primarily been done in the field of data streams with a focus on maintaining continuous up-to-date statistics with regards to various *k*, such as the median. Most of the research has concentrated on solutions that provide approximations instead of exact answers, primarily because of the following reasons: (i) data streams often tend to involve many tuples in short time periods whereby a certain degree of performance is required which cannot be achieved when exact results are required and (ii) exact results are most often not necessary to be able to use the statistical information for the desired purpose, which means approximations are more than acceptable in most use-cases. Subsection 2.1.1 discusses the different types of approximation techniques in further detail.

Research has also been done in the context of OLAP and data warehousing, although most of the focus of this research is into improving the performance of distributive and algebraic aggregate functions with very limited research into holistic aggregate functions. Subsection 2.1.2 discusses the existing state of this research field.

2.1.1 Approximation techniques

Approximation techniques are most commonly used when computing aggregation statistics for data streams. Data streams are different from data warehouses and therefore require different query processing techniques. The most important difference between a regular database/data warehouse and a data stream is that (i) the complete data set should be regarded as unbounded, i.e. new data is continuously incoming [23], and (ii) often at a fixed interval (e.g. every second) with a high volume (e.g. 1000 new tuples per second) [14]. One of the implications of this is that it is impossible to pre-compute any type of index, higher-level view of the data or other types of additional data structures to improve query processing because the full data set is never known. However it is possible to maintain a separate online data structure for improved query processing however this data structure must be maintained (i.e. updated with new data tuples) during the processing of the data streams. This adds additional restrictions on any new algorithm.

To be able to achieve the performance that is necessary to process data streams and process the desired aggregation queries most query processing algorithms provide solutions for approximate answers. The various solutions, some of which will be discussed in section 2.2, can be divided into one of the following types:

Sampling Algorithms based on sampling use a much smaller subset of the actual data set to process aggregation queries. By using a much smaller subset the algorithms do not need to complete a full pass over the whole data set every time new data is added but only need to complete a pass over the subset. Sampling can be random or deterministic. The method of sampling tuples to generate the subset greatly influences the actual result of the aggregation queries and is therefore generally the most important part of any sampling algorithm. An often-made improvement to sampling techniques are the use of pre-computed samples which are continuously updated during processing of the data streams. By maintaining a pre-computed subset of tuples query results can be computed even faster.

Wavelets Algorithms based on wavelets involve the use of complex transformations to get a representation of the full data set which is much smaller and can then be used to process aggregation queries. The original data is no longer necessary to compute a result for an aggregation query but any answer will always be an approximate answer.

Although approximation techniques can never return an accurate result, often an indication of the error in the approximation is returned. This can either be a guarantee on the error of the answer, i.e. the approximated answer is within 5% of the exact answer or a guarantee on the probability of a correct answer, i.e. in 95% of the times the approximated answer is correct but it is not possible to know exactly when. Some algorithms are able to provide both guarantees, i.e. in 95% of the times the approximated answer is correct and when it is not then the error falls within a margin of 5%.

2.1.2 OLAP approaches

The previous section discussed the different types of approximation techniques which are most commonly used for data stream processing. However these same techniques (e.g. sampling) are often also applied in the field of data warehousing, OLAP and decision support systems (DSS's). However they can often be optimized even further since in most cases the full data set is known before hand (a priori) and updates are relatively infrequent, which generally means that the complexity of updates is significantly simplified whilst the query processing part of any algorithm can be further optimized.

Since the full extent of the dataset is known a priori and updates are quite infrequent the use of additional data structures are very common with algorithms for optimizing query performance for data warehouses. These additional data structures come in all kinds of forms, such as trees, hash tables, lists, etc, and are used by the query engine to be able to compute the answer for any aggregation query using the separate data structures instead of the full data set, thereby achieving a (sub-)linear performance.

Subsection 2.2.3 outlines some existing algorithms for optimizing aggregation queries in data warehouses.

2.2 Existing algorithms & techniques

This section will provide a survey of existing techniques and algorithms based on sampling for optimizing query processing for aggregate queries and in specific for computing *k*th smallest element queries on a large dataset. Most of the algorithms outlined in this section do not fit the requirements exactly because none of them are able to compute an exact result however they do provide a background.

Subsection 2.2.1 will first list some of the most prominent sampling algorithms. Then subsection 2.2.2 will discuss some techniques based on wavelets. Finally, subsection 2.2.3 will outline some algorithms which are specific tailored towards OLAP purposes.

2.2.1 Sampling algorithms

This subsection will discuss some sampling algorithms for optimizing aggregation query performance.

In [24] Manku, Rajagopalan & Lindsay (1998) describe a general framework for single-pass approximate quantile finding algorithms on large data sets and present several new algorithms based on this framework with guaranteed error bounds. Only one of the described algorithms uses sampling, to reduce the memory footprint of the algorithm. They conclude that their algorithms represent a clear trade-off in accuracy and the memory requirements, which poses a challenging choice for users.

In [25] Manku, Rajagopalan & Lindsay (1999) continue with their previous work from [24] and outline two improvements on their original sampling method from their previous paper: (i) previously the length of the data set had to be known to the algorithm beforehand, which is not very practical when dealing with streams of data. In this paper they introduce a new sampling scheme that does not exhibit this property and (ii) they introduce a new algorithm that is better optimized with regards to memory for estimating extreme quantiles (i.e. quantiles with a value in the top 1% of the elements).

In [17] Greenwald & Khanna (2001) present a new online algorithm for computing quantile summaries of large data sets, which also does not need knowledge of the size of the data set a priori. Their algorithm is based on a pre-computed sorted data structure which consists of a subset of the complete data set and is dynamically maintained during arrival of new data. At this moment the work by Greenwald & Khanna is considered to be one of the best algorithms for quantile summary approximations [12].

In [15] Gilbert, Kotidis, Muthukrishnan & Strauss (2002) introduce a new algorithm for dynamically computing quantiles whereby the algorithm observes all update operations and maintains a small-space sample representation of the full data set. The results for all quantile queries are then computed using the sample presentation with an accuracy to a user-specified precision. The main contribution of their new algorithm is the ability to handle delete operations, which is generally not considered in previous research because this is not a very common or frequent operation in most data stream contexts.

In [11] Cormode & Muthukrishnan (2005) offer a new algorithm which, similar to the work of Gilbert et al., is also able to handle deletions and also maintains a representation of the full data set using a different method resulting in a smaller size and therefore improved performance. At this moment this work is considered to be one of the best algorithms which is also able to handle deletions from the data set.

2.2.2 Wavelet-based algorithms

This subsection will highlight some well-known algorithms that are based on wavelets which are able to provide an approximate result for aggregation queries. As with the algorithm from the previous subsection these works are not directly suitable for the use-case of this research but they do provide some insight into the existing work and related possibilities.

In [32] Vitter, Wang & Iyer (1998) present a technique based on multiresolution wavelet decomposition to get an approximate and space-efficient representation of the data cube, which is used in OLAP operations. Their technique results in approximate results for online range-sum queries with limited space usage. Unfortunately their approach is currently only suitable for sum aggregations with no clear way of applying it for kth smallest element queries.

In [6] Chakrabarti, Garofalakis, Rastogi, & Shim (2001) propose the use of multi-dimensional wavelets for general-purpose approximate query processing (as oppossed to an algorithm that is suitable for a very specific aggregation query). Their approach uses wavelet-coefficient synopses of the full data set and uses these synopses to compute an approximate result for queries, enabling very fast response times. Their results indicate that their approach provides a better quality approximation than sampling approaches.

In [14] Gilbert, Kotidis, Muthukrishnan & Strauss (2001) present techniques for computing small-space representations of large data streams using wavelet-based transformations, similar to the previous works. Their work also shows that a high-quality approximation can be computed for aggregation queries using wavelet-based representations of the full data set.

2.2.3 OLAP algorithms

This subsection will provide an overview of existing algorithms which try to optimize (aggregation) queries in an OLAP context. Whereas most algorithms mentioned in the previous two subsections are primarily directed towards data stream processing the algorithms reviewed in this subsection are very much geared towards data warehousing, whereby the data set set has (i) a fixed size, which is known beforehand and (ii) has relatively few updates or in batches and finally (iii) is often multi-dimensional, which is not the case for most data streams.

In [16] Gray et al (1997) provide a global description of the data cube, which has become an essential part of the OLAP landscape. Although they do not provide any specific algorithms for aggregation queries in this paper it is still relevant, since it is very likely that the work done as part of the research described in this thesis will involve the use of data cubes. In this paper Gray et al explain what the data cube is, how its fits in SQL and how it can be efficiently computed.

In [19] Ho et al (1997) present fast algorithms for range queries two types of aggregation operations: SUM and MAX. Range queries are queries whereby a range is specified for a certain dimension (e.g. time) and the aggregation is applied to the values within that range. This is exactly the goal of the research of this thesis for holistic aggregation functions, such as the median, instead of distributive functions as in this paper by Ho et al. Their approach is based on pre-computed sub-aggregate values stored in a hierarchical tree structure to be able to compute the result for any aggregate query. Unfortunately this approach is not (directly) suitable for the research described in this thesis because holistic aggregation functions cannot be calculated in a distributed manner.

In [13] Gibbons & Matias (1998) introduce two new sampling-based techniques, concise samples and counting samples, for computing approximate results for aggregation queries on data warehouses and present techniques for maintaining the samples as new data is inserted into the data warehouse. The main contribution of their work is the improvement in the accuracy of the computed results based on the new sampling techniques.

In [28] Poosala & Ganti (1999) propose the use of multiple histograms to approximate the data cube and compute the results for aggregation queries approximately using the summarized data (i.e. the histograms) instead of the data cube itself, which results in approximate results but a much lower response time.

In [7] Chaudhuri, Das & Narasayya (2001) present a solution based on sampling to approximately answer aggregation queries. Their main contribution lies in identifying the most optimal samples based on the (expected) query workload and they show how it can be implemented in a database system (Microsoft SQL Server 2000). Although their work is not directly applicable for use-case of this thesis, their approach to workload analysis and ”lifting” workload to improve performance does provide some interesting solutions.

In [8] Chiou & Sieg present an approach for optimizing query plans for holistic aggregate queries. Their work is very different from the previously seen research in that this work is actually quite fundamental and provides a very broad optimization for improving query performance for any holistic aggregate query. Although their approach does make it possible for query optimizers to generate improved query plans it’s unlikely that this solution will make it possible to drastically improve the performance of holistic queries in such as way that large data sets can be explored in real-time. Therefore their work might be considered an additional improvement upon another algorithm.

In [22] Li, Cong & Wang extend the original work done by Lakshmanan, Pei & Han (2002) in [21] by adding support for the median aggregation function to the Quotient Cube technique, which is a method for summarizing the semantics of a data cube in smaller size and is used to improve the performance of aggregate queries. Unfortunately the work done by Li, Cong & Wang is mostly focused on maintaining the quotient cube whereas no attention is paid to the query performance improvement.

In [5] Braz et al outline some of the issues that are present when trying to compute an holistic aggregation query for a trajectory data warehouse. The holistic function they discuss is the presence function, which returns the number of distinct trajectories in a particular spatio-temporal area. Their solution is to replace the actual holistic function with two distributive aggregate functions which approximate the actual function.

In [27] Nandi et al look at using the MapReduce framework to compute the data cube for holistic measures. Their approach is based on identifying a subset of holistic measures that are participially algebraic and use this property to make it easier to compute them in parallel. Unfortunately the class of holistic functions investigated in this thesis are not partially algebraic and therefore the work of Nandi et al is not directly applicable however their use of the MapReduce framework is something worth considering.

The next section will discuss existing techniques used by various RDBMS products to compute the results of median aggregation queries.

2.3 Approaches used by existing RDBMS products

This section will outline existing techniques used by several open-source and commercial database software products to compute the results of median aggregation queries to determine the current state-of-the-art algorithms for handling holistic aggregation queries. The following RDBMS products have been investigated:

- PostgreSQL (9.1)
- MonetDB (Jan2014 release)
- MySQL (5.1.23)
- SQL Server (Express 2014)

The reason these products have been chosen is that they are some of the most popular database products of the world and are continuously developed and improved upon. The exception to this is MonetDB, but this database has been included in the investigation because it is an very high-performance database often used in OLAP contexts and therefore relevant.

The next subsections will shortly discuss the existing approach used by each database.

2.3.1 PostgreSQL

PostgreSQL, as of version 9.3, does not come with a native implementation of the median aggregation function or any *k*th element aggregation function. However there are some community-based user defined functions that add this functionality to PostgreSQL. The best version is most likely the C-based implementation by Dean Rasheed (2010) [29], as it is the most complete user-written median function and offers the best performance. The algorithm behind this implementation is very simple; to calculate the median of a set of tuples, all the tuples are first collected and stored in memory. Once all the tuples have been collected they are sorted, counted, and then the median is collected by taking the middle element(s). Clearly this is the most basic algorithm of finding the median and is not likely to offer very good performance, especially as the data set increases. Since this approach is based on quicksort its time complexity is `O(nlogn)` and its memory complexity is `O(logn)`.

2.3.2 MonetDB

MonetDB has a native implementation of the median (or any other quantile) aggregation function and the exact implementation is publicly available [1]. This implementation is very similar to the user-created implementation for PostgreSQL since it also relies on counting the number of tuples, sorting them and then returning the middle tuple(s). The possible advantage of MonetDB is that it largely keeps its (active) data in memory thereby avoiding the need to scan large blocks of tuples from disk. Theoretically it has the same time and memory complexity as the implementation of PostgreSQL, however it is very likely that its actual time complexity is much lower (due to the fact that data is read from memory instead of disk) and its actual memory requirements much higher.

2.3.3 MySQL

MySQL also does not come with a native implementation of the median aggregation function thereby requiring the use of a user-defined function. Several different versions are publicly available and a C-based version [2] has been investigated. The algorithm used by this version is almost identical to algorithm used by the UDF for PostgreSQL and the native implementation of MonetDB; all the tuples are first collected and stored in memory. Then the list of tuples is sorted using quicksort and the middle tuple(s) is returned as the median. Clearly the time complexity and memory complexity of this implementation is exactly the same as the complexities of the PostgreSQL UDF.

2.3.4 SQL Server

SQL Server comes with a native implementation of the median aggregation function called the PERCENTILE_DISC function, which can be used to compute any percentile for a set of tuples. According to the documentation [26] this function first sorts the values and then returns the *k*th smallest element.

[1] MonetDB source code: http://dev.monetdb.org/hg/MonetDB/file/facde0cda494/gdk/gdk_aggr.c#l2189

[2] MySQL C-based median UDF, available at http://mysql-udf.sourceforge.net/

Based on this description it can be concluded that the algorithm behind this function is similar to all the other implementations discussed in the previous subsections. Given this information it is safe to assume that this function has a similar time and memory complexity as the other implementations.

2.3.5 Conclusion

This section investigated existing approaches used by several open-source and commercial RDBMS products used to process median aggregation queries. Two of the databases investigated did not have a native implementation to process median aggregation queries requiring the use of user-defined implementations. In both cases several different existing user implementations were available but only the C-based versions were investigated. The other two databases did have a native implementation to process median aggregation queries.

All implementations that have been investigated are based on the same algorithm whereby all the data is first sorted, counted and only then can the exact median value be retrieved. This algorithm is the most basic algorithm that is available and is not optimized in any way. It is therefore quite apparent that existing approaches do not offer the performance that is necessary for real-time exploration.

The next section will outline the current research done as part of Work Package 5 and its relevance.

2.4 Current research of COMMIT/Work Package 5

This section will outline the current progress of the research done in the context of Work Package 5 (WP5) [10] and discuss its relevance. WP5 is part of the TimeTrails sub-project of the COMMIT research community, focused on spatiotemporal data warehouses for trajectory exploitation. The objectives of Work Package 5 are as follows:

> Design and develop trajectory-based stream storage and query processing techniques to support real-time trajectory exploration and mining algorithms.

The research described in this thesis will primarily make a contribution to the second part of the objectives: designing and developing query processing techniques to suport real time [trajectory] exploration and mining algorithms. Additionally this thesis will not focus on stream-based purposes but on OLAP-based goals for decision support systems and offline analysis.

Current research at the University of Twente as part of WP5 has produced a new algorithm, called the Stairwalker algorithm, to optimize query performance for aggregation queries that are based on a distributive or algebraic function. In subsection 2.4.1 the Stairwalker algorithm is explained in further detail. With the Stairwalker algorithm and custom extensions to the Geoserver software[3] and the PostgreSQL RDBMS software[4] a prototype has been developed that allows real-time exploration of a dataset of millions of Twitter messages overlaid over a map with their geolocations. However the Stairwalker algorithm is unfit for holistic aggregation functions hence the reason for the research of this thesis: to develop a similar algorithm to the Stairwalker algorithm which is able to accelerate holistic aggregation functions.

The next subsection will discuss the Stairwalker algorithm in further technical detail.

2.4.1 Stairwalker algorithm

The Stairwalker algorithm, developed by van Keulen, Wombacher & Flokstra, is an algorithm that consists of several distinct parts that are used together to drastically increase the performance of aggregation queries with a distributive or algebraic function. It consists of the following two important elements:

[3]GeoServer: available at http://geoserver.org/display/GEOS/Welcome.
[4]PostgreSQL: available at http://www.postgresql.org/

Pre-aggregate structures

The pre-aggregate structures are supplementary data structures which are pre-computed a priory using the full data set and a specifically written Java tool for this purpose. Each pre-aggregate structure groups the full dataset into a smaller number of sub-aggregate values. Consider the sample data set shown in table 2.1.

time	1	2	3	4	5	6	7	8
value	9	8	7	10	6	1	4	5

Table 2.1: Sample dataset (level 0)

Using the Java tool various pre-aggregate subsets can be generated for the sample dataset for a specific aggregation function. Tables 2.2 and 2.3 show two different pre-aggregated subsets specifically suitable for the `sum` function.

time	1-2	3-4	5-6	7-8
value	17	17	7	9

Table 2.2: Pre-aggregate level 1 of the sample dataset

time	1-4	5-8
value	44	16

Table 2.3: Pre-aggregate level 2 of the sample dataset

In the first pre-aggregate, listed in table 2.2, tuples 1-2, 3-4, 5-6 and 7-8 have been summed up and the aggregated values are stored. In the second pre-aggregate structure, shown in table 2.3, tuples 1-4 and 5-8 have been summed and the aggregated value stored. The pre-aggregate structures are used by the query engine, as discussed in the next subsection, to compute the results for any aggregation query in the most efficient way.

Note that each pre-aggregate structure is only suitable for a specific aggregation function. In the example the pre-aggregates can only be used to optimize queries with the `sum` function and are not suitable for any other distributive or algebraic function.

Query engine

The query engine uses the pre-aggregate structures together with a complex algorithm to compute the results of a query as efficiently as possible. Consider the example of the previous subsection and the following (pseudo)query: `give the sum of tuples 1-5`. Without any pre-aggregates the query engine would have to retrieve, possibly from disk, tuples 1 through 5 and add them together to compute the result. However by using the pre-aggregates the query engine can take the first tuple of pre-aggregate level 2 (which represents the sum of tuples 1-4) and then retrieve tuple 5 from the original dataset (level 0) and add the two together, thereby only needing 2 retrievals and additions.

The complexity lies in trying to find the most optimal way of computing the result for a query when there are several different pre-aggregate levels. This complexity increases significantly for multi-dimensional data, which is most often the case when dealing with (large) data warehouses and OLAP. In the example the dataset is aligned along a single dimension but multiple dimensions changes the pre-aggregate structures into multi-dimensional data structures, which increases the cardinality of possible ways of computing the result for a query by multiple orders.

Calculating the possible ways of computing the result of a query has been implemented as a C extension for the PostgreSQL RDBMS software, which has been empirically shown to offer the best performance.

2.4.2 Protoype

As part of the research done at the University of Twente a prototype has also been developed that demonstrates the ability of the Stairwalker algorithm to facilitate real-time data exploration. This prototype consists of a web-based geographical map, served by the Geoserver software, overlaid with a grid whereby each grid cell contains the aggregation value for that part of the current geographical view, as seen in figure 2.1.

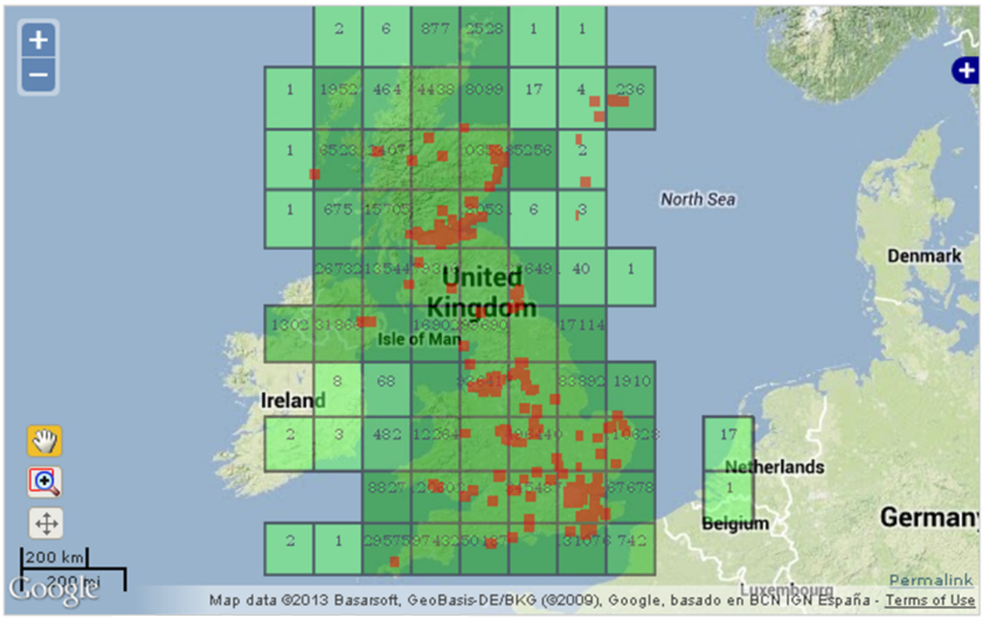

North Sea
Denmark
United Kingdom
Isle of Man
Ireland
Netherlands
Belgium
German
200 km
200 mi
Permalink
Map data ©2013 Basarsoft, GeoBasis-DE/BKG (©2009), Google, basado en BCN IGN España - Terms of Use

Figure 2.1: A screenshot of the Stairwalker prototype

The data for this overlay is dynamically computed using the appropriate aggregation query and is recalculated as the user zooms in/out or pans the map. A custom extension for the Geoserver software has been developed that generates the correct aggregation query, sends it to the database and transforms the database results into the correct grid layer.

2.5 Conclusion

This chapter provided an overview of the related work that is relevant to the research outlined in this thesis. First, this chapter discussed which existing research fields there are and the types of techniques that are currently being researched. Most of the research in the past 20 years has focused on approximation instead of exact answers, primarily to achieve the desired performance when dealing with large data sets or data streams.

There are several different types of approximation techniques, with sampling being the most common and most often researched. However there are many different types of sampling techniques, such as randomized sampling, determined sampling, online sampling, offline sampling and more. Sampling is most often used in conjunction with data streams however some OLAP approaches also tend to incorporate sampling to improve performance.

Another technique for approximation are wavelet transformations whereby the original dataset is transformed into representations, which are then used to compute the results for aggregation queries. These representations are small and therefore faster to process and store which results in improved query performance. However they can never be used for exact results because this cannot be determined with the wavelet representations.

In the field of data warehousing and OLAP the use of data cubes is very common. However data cubes cannot be used to accelerate aggregation queries with a holistic function because these cannot be computed in a distributive fashion, which is the basis for the data cube. This chapter outlined some existing research approaches that try to improve upon the data cube to enable support for holistic functions, albeit often in approximated or inefficient ways.

An investigation has also been into existing approaches used by several open-source and commercial databases to determine what kind of algorithms are used in practice to process median aggregation queries. Some of the databases investigated did not have a native implementation, requiring the use of external C extensions. In all implementations, native or user-based, the same basic algorithm lies at the heart of each approach. This algorithm simply sorts all the tuples, counts the total number of tuples and returns the middle tuple(s) as the median. Unfortunately this algorithm is unlikely to deliver the performance that is necessary for any type of real-time exploration.

Finally, this chapter outlined the current research progress at the University of Twente in the context of Work Package 5 of the TimeTrails/COMMIT project, which at this moment has resulted in the Stairwalker algorithm. This algorithm is able to greatly improve the performance of aggregation queries with an distributive or algebraic function, making it possible to explore large datasets in real-time, which has been demonstrated using a geographical prototype. Unfortunately this algorithm is unsuitable for aggregation queries with a holistic function and therefore a different approach is needed.

Clearly there already has been a lot of research in the field of holistic aggregation queries, query performance and large data sets. However most of the research has been focused on computing approximated results (with a certain error bound) in the shortest possible time. Whilst many of the techniques are semi-relevant for the research of this thesis it is not directly applicable because the focus of this research is on computing exact results.

The next chapter will discuss what factors influence the performance of an (holistic) aggregation query and how these factors can be measured.

Chapter 3

Query Performance

This chapter analyzes the factors that influence the performance of a database query and how these can be measured. It is important to have a clear understanding of exactly what factors influence query performance before it is possible to develop an algorithm that will improve these factors.

The first section of this chapter will review existing research and literature to determine exactly what the current knowledge is on factors that influence query performance. The second section of this chapter will provide measures for each performance factor based on existing research. The third section will discuss the empirical investigation into the factors that are believed to influence the performance of a query to verify that performance is really influenced as expected. Finally this chapter will end with a conclusion.

3.1 Performance factors

This section outlines the existing research into factors that affect the performance of database queries with a focus on OLAP and data warehousing. The most common database operation in this context are range queries that need to scan a majority part of the data set to compute the results for a query, most often in aggregated form [18]. Therefore focus will be on the factors that influence the performance of these types of queries, i.e. large read-only sequential scans.

The biggest factor that influences the performance of large sequantial scans of the data set are the number of bytes read of the hard disk where the data is stored [18]. Therefore most algorithms that focus on improving query performance are focused on reducing the number of bytes read of disk by employing several strategies, such as compression, employing the use of indices, reducing the amount of data that is needed to compute the final result, improving data layout, etc. There are several more factors that have an influence on the performance of database queries, however their influence is very small when compared to the number of bytes read from disk. Some other factors that can have an influence are: CPU clock speed, CPU instructions-per-cycle efficiency [4], CPU data caches and CPU bus speed [1].

Since disk access is the biggest factor that affects query performance this is discussed in further detail in the next subsection.

3.1.1 Disk access

Disk access, or more specifically the number of bytes read from the disk, is the biggest factor for query performance. There are two very specific aspects of disk access that can greatly influence the performance of a query: disk bandwidth and seek time. Each are discussed in the following subsections.

Disk bandwidth

Disk bandwidth is an indication of the number of bytes that can be read from the disk per second. Most modern hard disks are able, typically in a RAID-setup, to achieve a bandwidth of 300+ MB/s. Modern

Solid-State-Drive hard disks are able to achieve an even higher speed. For large sequential scans of data the bandwidth of a disk is an indication of the expected time to scan all the data from disk, e.g. a dataset of 10 GB will take to at least 34 seconds to be scanned once with a bandwidth of 300 MB/s. This immediately shows why an algorithm that improves performance is needed for real-time data exploration.

Seek time

Seek time is the time that a hard drive needs to find the correct location for a piece of (requested) data and physically move the head of the hard drive to the correct location, and usually takes between 5-10 ms [18]. This seeking is only required when data is required that is located on a physically different part of the hard disk from the previously requested data. This is the main reason why the location of the data is very important and can greatly influence the performance of a query. When a sequential scan is executed the seek time is generally negligible, if and when the scan is aligned along the sort order of the data. If the sequence that is requested is different from the order of the data then each individual tuple is randomly retrieved from the hard disk which in turn can lead to a huge cost in seek time. This can be illustrated with a quick example: for a dataset of 100.000 tuples and a seek time of approximately 7 ms the total seek time could be more than 1750 s if the complete dataset is scanned in random order. It is clear that the seek time is a very important factor that must be considered. It is worth noting that modern SSD hard disks have a seek time of less than 0.1 ms, thereby greatly reducing the impact of seek time on the performance of database queries albeit not completely.

3.2 Measuring performance

This section will discuss the existing methods that are available for measuring query performance, partly based on the factors that have been outlined in the previous section. First, several methods will be discussed that are used to measure the disk access in terms such as bytes read, number of disk accesses and seek times. Then, some methods will be examined that are used to measure the complete performance of a database query, i.e. not only the disk access but complete result computation. This section will end with an overview of the methods that have used in this research and the reasons for choosing them.

3.2.1 Measuring disk access

Measuring the disk access of a query and its effect on the performance is actually a tricky process and involves decomposing every step in the process of query execution. The only way to make this possible is with the use of additional tools that are available for each specific RDBMS. One example of this is the TKPROF tool, which is used to profile queries executed against an Oracle database which includes information on the disk cost of the query in terms of bytes read, time spent on disk operations, etc [30]. Another example of a profiling tool for a specific RDBMS is the Query Analyze tool for SQL Server [1].

Most RDBMS products also support the EXPLAIN SQL operation, which can be prefixed to any database query and returns the query plan which will be used by the database to compute the result for the query. Often, this query place also includes some indication of the cost of the query, usually in the form of the number of reads (in blocks or pages) from the hard disk (or in-memory buffer). Unfortunately this cost indication is often an approximate guess by the query engine of the database. Furthermore, cost of CPU time are often also included in this cost factor, making it impossible to accurately measure the true cost of the disk usage.

Another possibility is to monitor the performance of the disk usage at a lower level outside of the RDBMS, for example with a utility like ”iotop” [2]. This utility is able to provide much detail on the disk usage for a specific system process and can therefore be used to monitor the disk usage of the database process.

A big downside of measuring the disk performance of a query is that it is very schema and database type dependent thereby making any cross-schema or cross-database comparisons impossible. The number

[1] Overview of SQL Query Analyzer: http://technet.microsoft.com/en-us/library/aa216945(v=sql.80).aspx
[2] Iotop: http://guichaz.free.fr/iotop/

of bytes read varies greatly on the underlying data and also depends on how a database stores it data (i.e. compressed or not). Therefore any measures on raw disk usage are only useful for comparisons within a single dataset and database.

3.2.2 Measuring query performance

Measuring the overall query performance is generally performed by timing the execution time of a query, i.e. the time it takes to compute the results of a query [3]. This can be done using a black-box approach or a white-box approach. With the black-box approach an external application is used to time the execution how a query whereby the timer is started as soon as the query is sent to the database and the timer is stopped when the results are returned. However this approach suffers from the drawback that other factors, such as network speed or client slowdowns, can affect the performance of the query even though the database engine has no relation with these factors.

Therefore a white-box approach is likely to provide more accurate measurements than a black-box approach. In a white-box approach the execution time is measured by the database itself thereby removing any outside factors, such as network speed. Unfortunately, not all RDBMS products support this functionality which means a black-box approach might be necessary for some types of database.

3.2.3 Choice of methods

The previous subsections have outlined several methods that are available for measuring the performance of database queries ranging from measures that indicate a very specific performance metric, e.g. disk bandwidth, to measures that give an indication of the overall performance of a query. In this research the following methods have been employed to determine the performance of the test set of aggregation queries:

White-box measuring of overall query performance
The overall performance of the aggregation queries has been measured using a white-box approach, because that's what is most important for the goal of this research - real-time data exploration - and must therefore ultimately be improved. It's not entirely unlikely that disk usage might be improved but overall query performance actually decreases due to a massive increase in CPU time or another performance factor. Preference goes to white-box measuring because that provides the most accurate results however this might not possible for every database, in which case black-box measuring will be used as a fall-back.

Given the results of the investigation into the query performance factors, this measure is sufficient enough to be able to properly asses the performance of any query, and therefore makes it possible to determine if the developed algorithms are able to improve the performance of holistic aggregation queries.

Specific IO measures to get insight into disk usage performance have not been chosen because it is too unreliable for two reasons: (i) disk IO might vary greatly between multiple executions of the same query due to caching by the database itself, which is often difficult to detect and/or disable and (ii) these measures cannot be used for any type of comparison between different datasets or database types.

3.3 Empirical verification

This section describes the work that has been done as part of the empirical verification of the performance factors and measures discussed in the previous sections. The first subsection gives an overview of the test setup used. In the second subsection the results of the verification are shown and discussed in detail. Finally, in the last subsection of this section a conclusion of the empirical verification is given.

3.3.1 Test setup

The test setup for empirical verification consists of the following parts:

Datasets
Multiple datasets have been generated of a varying number of tuples ranging from 1000 tuples up to 10 million tuples, whereby each tuple consists of an unique id (as primary key) and a random 32-bit integer. Each dataset is ordered on its primary key in ascending order.

Test queries
A very simple aggregation query is used for verification which performs a sum over all the tuples in the data set, as can be seen in listing 3.1. To compute the result for this query all the tuples in the dataset must be processed, either from disk or in-memory cache.

```sql
SELECT SUM(value) FROM dataset;
```

Listing 3.1: Test query used for verification of performance measures

Hardware
Verification has been done on a laptop computer with 8GB of memory, a dual-core CPU of 2.2 GHz and running Windows 7 64-bit.

Database software
Verification has been done with three different database software products: MonetDB, PostgreSQL and MySQL. These three products have been chosen for their availability (all three are open-source) or their suitability for OLAP purposes (MonetDB).

With these components empirical verification has been performed by executing each test query using a client-side command-line database tool and taking the measurements as described in subsection 3.2.3. The results of this verification are discussed in the next subsection.

3.3.2 Results

This subsection presents the results of the empirical verification of the performance measures to verify that they are suitable for this research.

# of tuples	Execution time (ms)
1,000	0.18
10,000	1.50
100,000	15.00
1,000,000	155.00
10,000,000	1,775.00

Table 3.1: Results for empirical verification of performance measures with PostgreSQL

Table 3.1 lists the results of the empirical verification with PostgreSQL. The results show that the execution time measure is a good indication for overall query performance; as the dataset increases in size with a certain factor the execution time increases with approximately the same factor. This means that the computational complexity of the query is quite accurately indicated by the execution time.

# of tuples	Execution time (ms)
1,000	0.66
10,000	3.76
100,000	44.03
1,000,000	342.11
10,000,000	5,818.33

Table 3.2: Results for empirical verification of performance measures with MySQL

In table 3.2 the results of the empirical verification with MySQL are shown. These results show that again the execution time is a good measure for an indication of the overall query performance. The execution time is measure in a white-box approach, i.e. by MySQL itself as opposed to an 3rd-party client. This is done with the use of MySQL's profiling functionality and its usage is demonstrated in listing 3.2.

```
SET profiling = 1; # turn on profiling
SELECT SUM(value) FROM dataset; # execute test query
SHOW PROFILES; # retrieve query profiles
SHOW PROFILE FOR QUERY 1; # returns specific profile information for query
```

Listing 3.2: Demonstration of the usage of MySQL's profiling functionality

# of tuples	Execution time (ms)
1,000	233.23
10,000	236.02
100,000	231.75
1,000,000	2,226.08
10,000,000	2,234.02

Table 3.3: Results for empirical verification of performance measures with MonetDB

Table 3.3 shows the results of the empirical verification with MonetDB. These results are quite different from the results of PostgreSQL and MySQL, because the relationship between the number of tuples and the execution time is not directly related. The execution remains constant for the first 100.000 tuples and then jumps drastically for the next measurement at 1 million tuples and then stays constant again. This is most likely due to an internal mechanism of MonetDB, however it does have an influence on the performance measure making it less reliable. Unfortunately MonetDB does not easily offer additional measures for overall performance although it is possible to profile queries in much greater detail using additional external tools, such as the stethoscope[3] and tomograph[4].

The execution time in MonetDB is measured in a white-box approach by enabling the query history in MonetDB and retrieving information on the latest query, as in listing 3.3.

```
SET history=true; # turn on query history
SELECT SUM(value) FROM dataset; # execute test query
SELECT query, exec FROM querylog ORDER BY ctime DESC LIMIT 1; # fetch execution time for query
```

Listing 3.3: Demonstration of the usage of MonetDB's query history

3.3.3 Conclusion

This section has described the work that has been done in empirically verifying the query performance factors and related measures that have been discussed earlier in this chapter. As part of this verification a single test query has been executed against a dataset of varying size on multiple different types of databases. For each database type it has been verified whether execution time is an appropriate measure for query performance and looked at available additional type-specific measures.

Results have shown that the query response time is very suitable as an indicator of overall query performance for PostgreSQL and MySQL, because a direct relationship exists between the complexity of the query and the response time. For monetDB however the results did not conclusively show this and additional measures are needed.

In all cases it was possible to measure the response time for the query using a white-box approach using inbuilt functionality offered by each database. This means that the results do not suffer from any external influences, such as network speed.It is assumed that other commercial database products (e.g. Oracle, SQL Server) also offer such functionality.

[3]Stethoscope: http://www.monetdb.org/Documentation/Manuals/MonetDB/Profiler/Stethoscope
[4]Tomograph: http://www.monetdb.org/Documentation/Manuals/MonetDB/Profiler/tomograph

3.4 Conclusion

This chapter has analyzed the most important factors that influence the performance of a database (aggregation) query and how they can be measured. Research into existing literature into query performance has shown that the single most important factor that has a great influence on query performance is disk IO, which can be further specified in disk bandwidth and seek time. Disk bandwidth indicates how much data can be read per second (i.e. bytes/second) whereas seek time is the time a random read of a tuple takes, which can take up to 5-10 ms for a single seek. Hence, improving performance for holistic aggregate queries will most likely involve the reduction of disk IO.

This chapter has also discussed the methods that are available for measuring the performance factors, with a focus on two specific categories of measures: measuring disk IO and measuring overall query performance. Although some methods have been found for measuring disk IO in conjunction with database queries there are several downsides that make this a cumbersome process to use in this research, namely (i) disk IO measures are schema and database dependent, which makes any cross-schema or cross-database comparisons impossible and (ii) many IO measures are specific for a specific database product. To measure the overall performance of a query only one method has been found and that is to measure the execution time (in number of seconds) of the query until results are returned, although this can be done using two different approaches: a white-box approach, whereby the execution time is measured by the database itself, or a black-box approach, whereby the execution time is measured by an external client. The second approach suffers from external factors, such as network speed, and is therefore less reliable. However the white-box approach does require that a database is able to time its own queries.

Based on the research into the query performance factors and the methods that are available for measuring them a choice has been made to only measure the overall performance of aggregation queries, preferably with a white-box approach, because this measure is the only measure that offers reliable results and is sufficient enough to be able to properly asses the performance of any aggregation query.

This chapter ended with a description of the work done into empirically verifying the theoretical research into performance factors with a test data set of varying size using a simple test query and three different database types (PostgreSQL, MySQL and MonetDB). The results have shown that the execution time of a query has a direct relationship with its complexity for PostgreSQL and MySQL but for MonetDB the relationship is more complicated, which might necessitate additional database specific tools.

The next chapter will discuss the experiments setup used in this research to test various algorithms.

Chapter 4

Experimental setup

This chapter describes the exact setup used in this research to determine the performance of the default basic median algorithm and the alternative developed algorithms. Section 4.1 outlines the datasets that have been used for in the research and their unique characteristics. Then, section 4.2 discusses what type of queries have been used to test algorithms and how these test queries have been created. Next, section 4.3 describes the details of the platform which was used to run experiments on in terms of hardware and software. Finally section 4.4 gives a detailed description of how the experiments are actually run.

4.1 Datasets

In this research 3 different types of datasets have been used, each with different properties and data distribution. All 3 datasets consist of a single table with multiple columns whereby some columns are the dimensions of data and one or more columns are the actual measures. Table 4.1 gives an overview of the three datasets used in this research.

Dataset name	**Number of dimensions**	**Measure**	**Number of tuples**
PEGEL	1: time	Waterheight (in cm) of the river Thur, near Andelfingen	± 118,000
KNMI	2: time and station	Temperature (in celcius) at specific station locations in the Netherlands	± 12,500,000
Twitter	3: time and location (x,y)	Text length of Dutch geo-tagged Tweets, collected by the UTwente	± 15,000,000

Table 4.1: Overview of the three datasets used in our reserach

The next subsections will discuss each dataset in more detail.

4.1.1 *PEGEL* dataset

The PEGEL dataset consists of approximately 118,000 tuples with a single dimension and a single measure. This dataset comes from a sensor station in Andelfingen, Switserland, as seen in figure 4.1, which measures the height of the river Thur every 10 minutes. Table 4.2 shows a small sample of this dataset. Since the dataset only has around 118,000 tuples the dataset has been duplicated multiple times to get a dataset of up to 10 million tuples. This duplication has been done by increasing the dimension (time) with the same interval as the rest of the dataset whilst taking an (existing) random value for the measure (waterheight).

Figure 4.1: The Andelfingen sensor station near the river Thur

ID	Timestamp	Time/date	Waterheight (cm)
1	1167606600000	2007-01-01 00:10:00	354.851
2	1167607200000	2007-01-01 00:20:00	354.852
3	1167607800000	2007-01-01 00:30:00	354.852
4	1167608400000	2007-01-01 00:40:00	354.853
5	1167609000000	2007-01-01 00:50:00	354.854

Table 4.2: Sample of the PEGEL dataset

4.1.2 *KNMI* dataset

The KNMI dataset consists of approximately 12,5 million tuples with two dimensions and multiple measures, however in this research focus has been exclusively on a single measure, namely temperature. The two dimensions of this dataset are time, similar to the PEGEL dataset albeit with a different interval between each tuple, and station. The dataset consists of 10 distinct measuring stations, with each station taking its own measurements. Table 4.3 shows a small sample of this dataset. Since this dataset contains enough tuples no duplication has been performed. This dataset has been downloaded via the website of the Royal Netherlands Meteorological Institute (KNMI) [20].

ID	Station	Timestamp	Date/Time	Temperature
1	1	946684800	2000-01-01 01:00:00	58
2	1	946685400	2000-01-01 01:10:00	60
3	1	946686000	2000-01-01 01:20:00	61
1001	2	946684800	2000-01-01 01:00:00	71
1002	2	946685400	2000-01-01 01:10:00	-24
1003	2	946686000	2000-01-01 01:20:00	-51
2001	3	946684800	2000-01-01 01:00:00	54
2002	3	946685400	2000-01-01 01:10:00	54
2003	3	946686000	2000-01-01 01:20:00	57

Table 4.3: Sample of the KNMI dataset

4.1.3 *Twitter* dataset

The Twitter dataset has approximately 15 million tuples whereby each tuple has 3 dimensions, time, an x-coordinate and a y-coordinate and a single measure, the length of the Twitter message for each tuple.

The dataset has been collected by the Databases group of the University of Twente and is currently still growing. Table 4.4 shows a small sample of this dataset. This dataset also has enough tuples for this research therefore no duplication has been performed.

ID	Timestamp	Date/Time	X-coordinate	Y-coordinate	Length
430788266552737793	978350401	2001-01-01 13:00:01	5.78581153	51.75877801	87
430788269983662080	978350402	2001-01-01 13:00:02	4.75604476	51.58726959	19
430788270801567744	978350403	2001-01-01 13:00:03	4.43317543	51.30735477	54
430788270449229824	978350404	2001-01-01 13:00:04	4.88858049	52.38793447	28
430788272651264000	978350405	2001-01-01 13:00:05	5.80288886	50.93610432	41
430788273414631424	978350406	2001-01-01 13:00:06	5.84096519	51.84279646	59

Table 4.4: Sample of the Twitter dataset

4.1.4 Different sizes

For each of the previously discussed datasets several versions of different sizes have been created using the original dataset as the base dataset. The goal of this is to test the performance of various median algorithms for a different number of tuples to get a clear picture of the behavior of the algorithms. Table 4.5 lists the versions of the datasets that have been generated.

Name	Size (nr. of tuples)
10k	10,000
100k	100,000
500k	500,000
1mm	1,000,000
10mm	10,000,000

Table 4.5: The different versions generated for each dataset

The next section will discuss the queries used during the experiments and how they have been created.

4.2 Test queries

This section will outline the queries used in the experiments to determine the performance of the algorithms.

For each version of each dataset a set of 5 queries has been generated that cover a random range of a certain number of tuples within the dataset using a start and end value for the dimensions of that dataset. Each query has been generated specifically to cover an exact number of tuples. Table 4.6 shows a few test queries for the 10mm version of the *PEGEL* dataset.

QueryID	Start timestamp	End timestamp	Rows
1	5514262800000	5824312200000	500,000
2	6628554600000	6938625000000	500,000
3	6682747800000	6992793600000	500,000

Table 4.6: Test queries for the PEGEL dataset

For each version of each dataset the following sets of 5 queries have been generated, whereby the percentage indicates the percentage of rows of the total dataset that are covered by the query:

- 5%
- 25%
- 50%
- 75%
- 95%

To illustrate this percentages above, test queries with the following sizes have been generated for the 10k version of the *PEGEL* dataset: 500 rows, 2500 rows, 5000 rows, 7500 rows and 9500 rows.

The next section will discuss the test platform which was used to run the experiments on.

4.3 Test platform

This section will outline the details of the test platform used to run the experiments on. The primary test platform consists of a single server managed by the University of Twente. This server, commonly called *Silo1*, has the following specifications:

- Dual quad-code Intel Xeon @ 2.33 GHz processor
- 8 GB of working memory
- Running OpenSUSE 10.3 32-bit

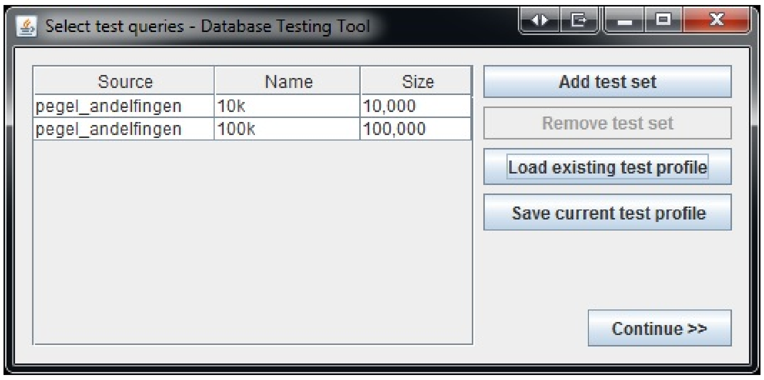
All the experiments have been done on this platform to ensure a stable environment between different experiments.

The primary database engine to run experiments with is PostgreSQL 9.1 with the PL/Java extension [31]. The main reason these products have been chosen is the fact that the PL/Java extension, which is only suitable for PostgreSQL 9.1, has made it possible to develop prototype implementations of algorithms with Java, which enables development at a very fast pace and provides an abstraction layer that removes much complexity when developing implementations for experiments.

The next section will describe how the experiments have actually been executed.

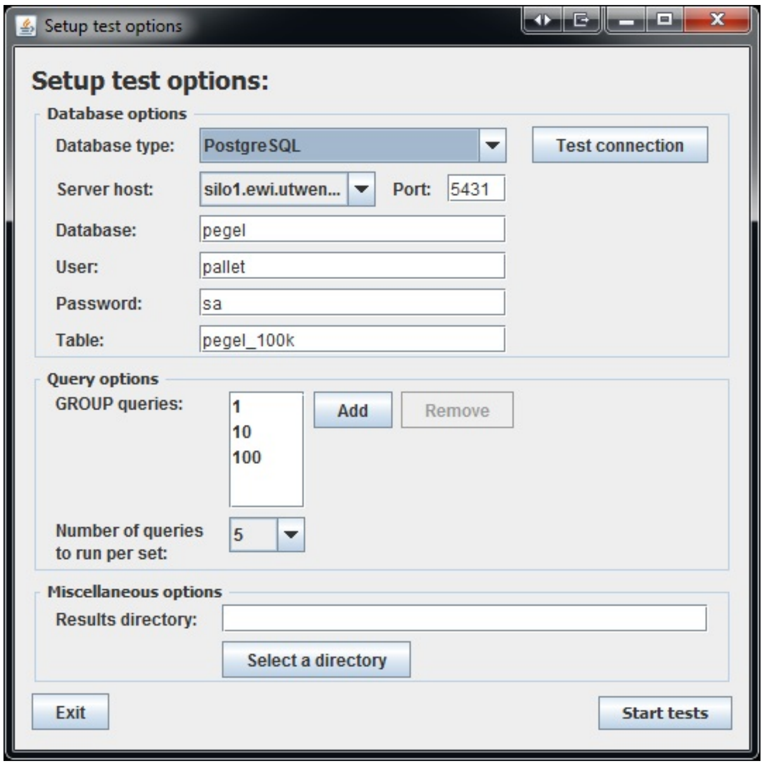
4.4 Running experiments

This section will describe how the experiments have been performed. To ensure that each experiment is run in exactly the same way a tool has been developed which runs a complete experiment automatically. This tool first requires an initial setup whereby the correct dataset and associated test queries are selected and the details of the database, as seen in figures 4.2 & 4.3. When the tool is setup the experiment can be started, at which point the tool will run all the test queries against the dataset and automatically records the execution time of each query as a measure of the query performance, as demonstrated in figure 4.4.


Select test queries - Database Testing Tool
Add test set
Remove test set
Load existing test profile
Save current test profile
Continue >>

Source	Name	Size
pegel_andelfingen	10k	10,000
pegel_andelfingen	100k	100,000

Figure 4.2: The first screen: to select the test queries to run



Setup test options:

Database options

Database type: PostgreSQL
Server host: silo1.ewi.utwen... Port: 5431
Database: pegel
User: pallet
Password: sa
Table: pegel_100k

Query options

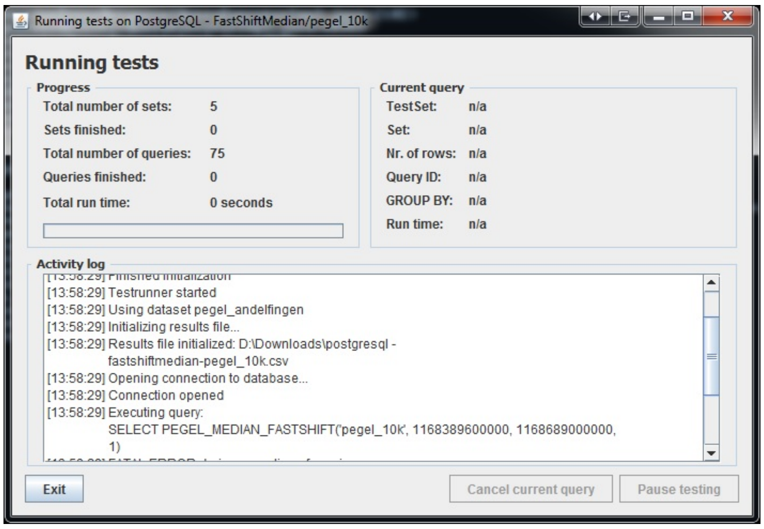
GROUP queries: 1, 10, 100

Number of queries to run per set: 5

Miscellaneous options

Results directory:

Figure 4.3: The second screen: where the database details and query options are set



Running tests

Progress

Total number of sets: 5
Sets finished: 0
Total number of queries: 75
Queries finished: 0
Total run time: 0 seconds

Current query

TestSet: n/a
Set: n/a
Nr. of rows: n/a
Query ID: n/a
GROUP BY: n/a
Run time: n/a

Activity log

```
[13:58:29] Finished initialization
[13:58:29] Testrunner started
[13:58:29] Using dataset pegel_andelfingen
[13:58:29] Initializing results file...
[13:58:29] Results file initialized: D:\Downloads\postgresql -
           fastshiftmedian-pegel_10k.csv
[13:58:29] Opening connection to database...
[13:58:29] Connection opened
[13:58:29] Executing query:
           SELECT PEGEL_MEDIAN_FASTSHIFT('pegel_10k', 1168389600000, 1168689000000,
           1)
```

Figure 4.4: The last screen: showing the progress of the tests as they are automatically executed

It is important to note some additional details with regards to the experiments. First of all, each test query is run 3 times with a different grouping configuration, since part of this research is to investigate aggregation functions in conjunction with a grouping operator. Therefore each test query is run as (i) a single group, (ii) divided into 10 groups and (iii) divided into 100 groups to determine if and how the performance of an algorithm changes depending on the grouping operator.

Secondly, during every experiment every step must be undertaken to ensure that caching, either by the RDBMS itself or the underlying operation system, does not influence the results of the experiment. Therefore the database server is restarted between each set of queries and each experiment to clear any caching that is done by the RDBMS itself. Additionally, the disk cache of the operation system is also emptied by executing the command in listing 4.1. This commands frees the pagecache, dentries and inodes and empirical verification confirms that this does indeed clear the disk cache.

```
sync \&\& echo 3 > /proc/sys/vm/drop_caches
```

Listing 4.1: Command-line command to clear the disk cache of the operation system

The next chapter will discuss the work that has been done into developing new algorithms for determining the median.

Chapter 5

Algorithm development

This chapter describes the work that has been done into developing a new algorithm for determining the median of a large dataset. This chapter will start by giving a clear description of the basic algorithm for finding the exact median, currently used by most database engines, discuss the query performance of this algorithm and its pros and cons. Then, several alternative algorithms will be discussed, each with different pros and cons with regards to query performance. Finally, this chapter will finish with a conclusion.

5.1 Basic algorithm

The basic algorithm for finding the median of a set of tuples, either a single set or multiple sets (in case of a group/aggregation query), is a very simple algorithm and is currently used by most database engines, as already described in section 2.2. To determine the median the algorithm performs the following steps:

1. Fetch all the tuples within the range of the query or current group from disk and store these in memory
2. Sort all the tuples in-memory using one of the many well-known sorting algorithms. Most commonly used is quicksort as it generally offers the best performance
3. After sorting, the median of the query or current group can now be directly returned by locating the middle element or elements from the in-memory sorted set

Clearly this is a very simple algorithm, requires no pre-computed data structures (such as additional tables or indexes) and is fairly data independent. However it is very likely that this algorithm is not able to offer the query performance that is needed for real-time data exploration and is likely to scale very poorly as the amount of data increases. As discussed in chapter 3 query performance is influenced by several factors, with disk I/O being the primary factor whereby two different types of disk I/O, seek time and bandwidth, can be differentiated.

For this algorithm *all* the tuples must be fetched from disk to be able to determine the median. As the dataset becomes bigger the number of tuples affected by queries also grows thereby increasing the number of tuples that must be fetched from disk. It is likely that these tuples can be fetched sequentially from disk, thereby avoiding the costly seek operations, but this still means disk bandwidth is a limiting factor for the query performance.

Another factor which likely has a negative impact on the query performance of this algorithm is the necessity of storing and sorting all the tuples in-memory. Although generally in-memory operations are regarded as very fast, especially when compared to disk operations, these cannot be ignored when dealing with hundreds of thousands operations to store and sort a large dataset in-memory. Additionally if the range of the current query or current group is very large, which is more likely with bigger datasets, it is likely that the number of tuples exceeds the amount of memory available to the database engine thereby necessitating the use of temporary tables on disk or memory swapping (on disk). This means that the query performance is again negatively influenced by disk I/O, possibly even increasing seek time.

From a theoretical standpoint therefore this basic algorithm has two main problems in determining the median of a (group-aggregation) query, which are:

1. *All* the tuples affected by the query must be fetched from disk
2. All the tuples for the current query or group must be stored and sorted in-memory

Using the experimental setup described in the previous chapter the query performance of this algorithm has been measured to determine if its performance matches the theory as described. Figure 5.1 shows the results of a C implementation of the basic algorithm with the 10MM version of PEGEL dataset. On the y-axis is the query execution (in seconds) as an inverse representation of the query performance and on the x-axis are the number of tuples affected by the query. The graph has 3 distinct series of measurements: a series whereby the query is grouped into a single group, a series whereby the query is grouped and aggregated into 10 groups and a series whereby the query is grouped and aggregated into 100 groups.

This graph shows a clear linear relationship between the query performance and the number of tuples affected by a query, i.e. as the number of tuples increases the actual query performance worsens. Based on this graph it is not clear what the underlying cause of this performance is, but it is likely that this performance behavior is caused by the disk I/O needed to fetch all the tuples from disk and storing and sorting them in-memory. These results suggest at this point there is no additional disk I/O caused by out-of-memory conditions, because this would have to be indicated by a very clear difference in the grouping series, which is not the case in this graph, i.e. all 3 series have almost identical query performance.

Surprisingly, the group-10 queries consistently have the worst performance of the 3 query grouping configurations. From a theoretical standpoint, the group-1 queries should have the worst performance, since these queries require *all* tuples involved in the query to be stored and sorted in-memory whereas with the group-10 and group-100 queries the tuples can be stored and sorted per-group. The exact reason for this is unclear but one possible reason could be that the logic behind the grouping mechanism of the database engine also has a certain performance overhead.

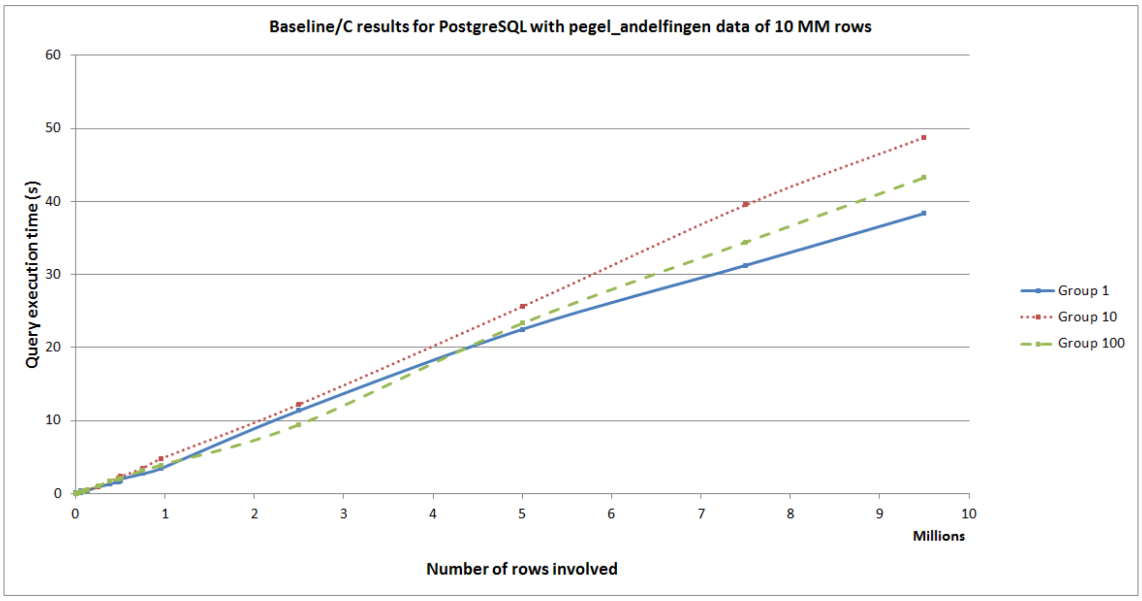


Figure 5.1: Results of the measurements with the basic algorithm and the 10MM PEGEL dataset

Two main theoretical problems have been identified with the basic algorithm and experimental measurements confirm that these problems do affect the query performance for real queries and data. Therefore an alternative algorithm is needed that does not suffer from these problems. The next section will discuss the work done into developing an algorithm whereby fetching all the tuples from disk is no longer necessary nor in-memory storage and sorting.

5.2 The shifting algorithm

The first alternative algorithm for determining the median, called *the shifting algorithm*, tries to solve the two main problems with the basic algorithm, as discussed in the previous section. The idea behind the shifting algorithm is that the median of a query or group is calculated by shifting the (pre-calculated) median of the whole dataset towards the median of the current query using (pre-calculated) shift factors. This section will first describe how this alternative algorithm works and is able to solve the problems of the basic algorithm. Then, the shortcomings of this algorithm are discussed and some actual measurements of the query performance with the test data sets are outlined.

The two main problems that have been identified with the basic algorithm were (i) the necessity of fetching *all* tuples from disk and (ii) the need to store and sort in-memory all the tuples to be able to determine the median. The shifting algorithm solves these problems by employing the following steps:

1. Calculate a priori the median of the whole dataset
2. Pre-compute, for each tuple in the dataset, the total number of tuples that are below the dataset media minus the total number of tuples above the dataset median and **before** the current tuple (dimension-wise). This is called the *left shift factor*.
3. Pre-compute, for each tuple in the dataset, the total number of tuples that are below the dataset median minus the total number of tuples above the dataset median and **after** the current tuple (dimension-wise). This is called the *right shift factor*.
4. Pre-compute a single binary stream of tuples that are below the dataset median and sorted by value (instead of by the dimension). This is called the *left data stream*.
5. Pre-compute a single binary stream of tuples that are above the dataset median, also sorted by value. This is called the *right data stream*.
6. To compute a range (aggregation) query the algorithm uses the left- and right shift factors together with the binary streams to shift the dataset median towards the median of the query eventually resulting in the exact median of the query.

With these key steps the shifting algorithm is able to exactly determine the median of a query or current group without needing to fetch all the tuples from disk, storing them in-memory or sorting them thereby solving the two main problems of the basic algorithm. This algorithm will be demonstrated with a simple example. Consider the sample data shown in table 5.1, which has a single dimension (time) and a single measure (value) and is sorted by its dimension.

time	1	2	3	4	5
value	9	8	7	10	6

Table 5.1: Sample dataset

Now consider the following query: give the median for all tuples that fall within the time range [1,4]. To be able to answer this query with the shifting algorithm the following steps have to be pre-computed a priori:

- The median of the whole dataset, which for the sample dataset is tuple with time=2 and value 8
- The left- and right shift factor of all the tuples in the dataset. These are listed in table 5.2.
- The left- and right data streams, as shown in figure 5.2. Only the binary data is actually stored and the decimal values are for reference only. Note that these numbers refer to the time ID of each tuple (and not the value). It's also important to see that the left data stream is sorted by the value of each tuple in a descending order and the right data stream is sorted by the value in an ascending order.

time	1	2	3	4	5
value	9	8	7	10	6
left shift factor	0	-1	-1	0	-1
right shift factor	+1	+1	0	+1	0

Table 5.2: Sample dataset with shift factors

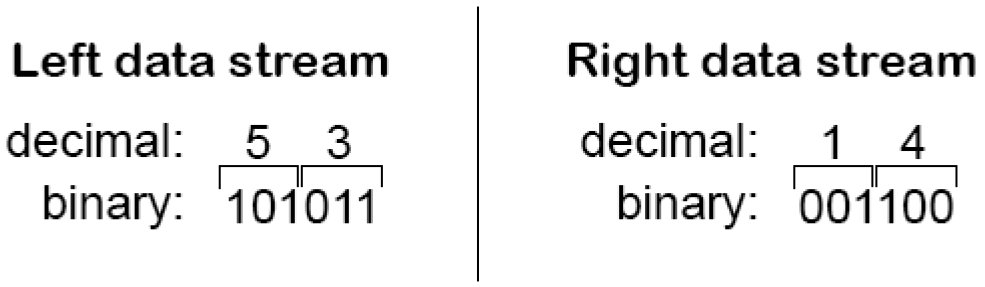


Figure 5.2: Pre-computed left- and right data streams for the sample dataset

Having pre-computed these additional data structures the example query can now be answered by the algorithm in the following way. First, calculate the query shift factor by adding the left shift factor of the start of the query time range and the right shift factor of the end of the query time range. For this query the left shift factor is 0 and the right shift factor is +1 making the query factor be +1. This means that the median of the whole dataset must be shifted 1 position to the right. At this point the right data stream is retrieved and used to scan 1 position to the right where tuple with time=1 is found. Since the number of tuples affected by this query is even the median must consist of the values of two tuples, which are the original dataset median (time=2) and the tuple 1 position to the right (time=1), which have values of 8 and 9 thereby making the median $\frac{8+9}{2} = 8.5$.

Since this algorithm no longer needs to fetch all the tuples from disk, thereby greatly limiting the amount of disk I/O operations, the query performance should, from a theoretical standpoint, scale significantly better as the number of involved tuples increases. Test measurements confirm that this is indeed the case, as is clearly shown in figure 5.3. This graph shows that with the shifting algorithm there is no longer a relationship between between the number of tuples affected by a query and the query performance. Instead, there is now a linear relationship between the number of groups and the query performance which is an indication that the actual shifting operation has a certain fixed cost per group thereby making queries with many groups more costly.

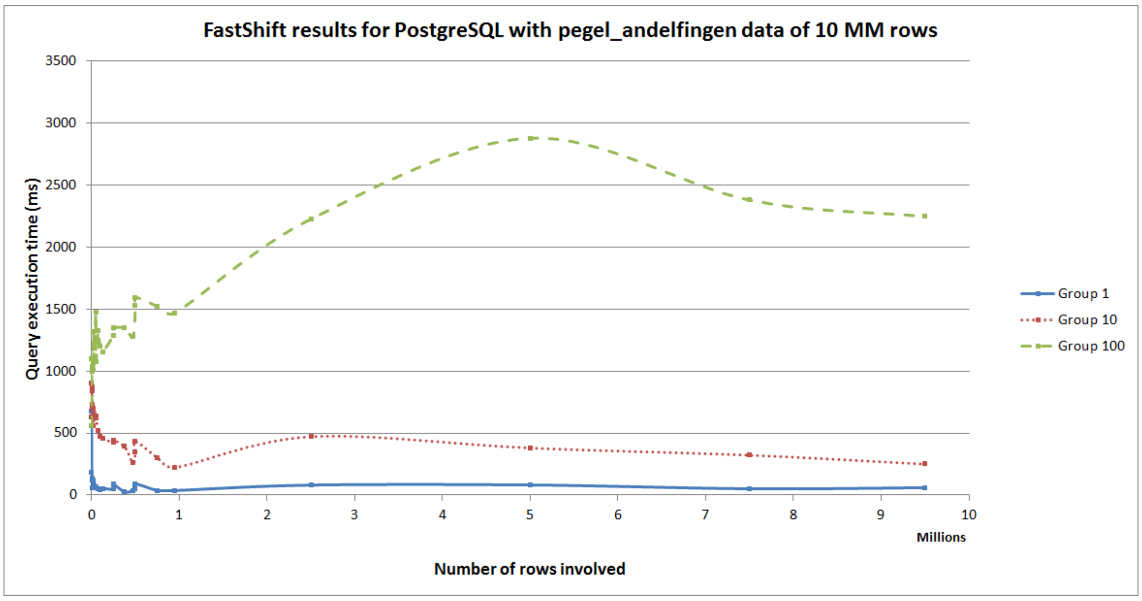


Figure 5.3: Results of the measurements with the shifting algorithm and the 10MM PEGEL dataset

Although the shifting algorithm solves the two main problems of the basic algorithm and completely changes the query performance profile it does have its own disadvantages. The biggest problem with the shifting algorithm is its limited suitability for higher dimensional datasets. The PEGEL dataset only has a single dimension (i.e. time) thereby making it simple to calculate the shift factor for a query, as explained in the key steps earlier in this section. However for a dataset with two or more dimensions the calculations necessary to determine the shift factor becomes increasingly more computationally expensive and complex.

Therefore an alternative algorithm is necessary for datasets with multiple dimensions. Such an algorithm will be discussed in the next section. But first another improvement on the shifting algorithm will be outlined that has the potential to even further reduce the amount of disk I/O. With the current version of the algorithm it is still necessary to fetch the actual measure value from disk (or disk buffer) once the correct tuple has identified as being the median tuple. What if this could be avoided? For a query with a single group this would likely have a negligible effect however for queries with many groups, such as the group-100, this could avoid much disk seek time.

To be able to circumvent these (possibly) costly disk fetches the actual measure value is included in the left- and right data streams instead of only the tuple key. This has the potential to increase the size of the data streams, especially for complex values which take up a lot of size, however prevents the potentially costly disk fetches. Actual test results have shown that this method of including the values into the data streams slightly improves the query performance, although not nearly as much as expected.

Figure 5.4 shows the test results with the shifting algorithm and values included for the PEGEL dataset. These results show that including the values can have a significant impact on the query performance (up to 2x as fast) however this depends very much on the number of rows involved and the number of groups, i.e. the effect is much larger for the group-100 queries than the group-1 queries, as seen in the figure. Based on these results it seems likely that as the number of groups increases the performance improvement also becomes bigger.

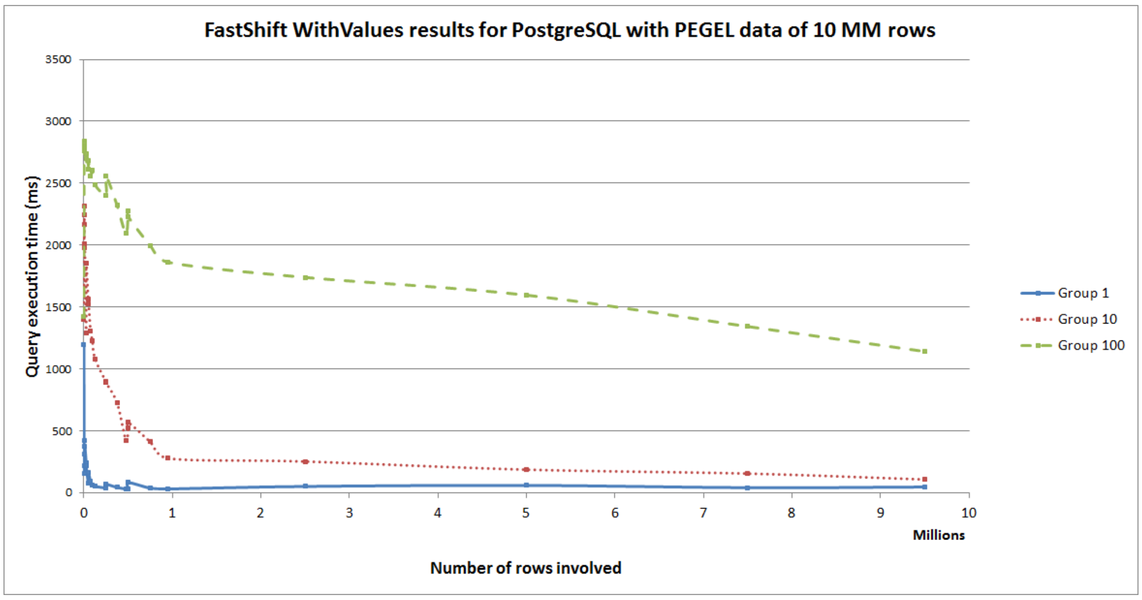


Figure 5.4: Results of the shifting algorithm with values and the 10MM PEGEL dataset

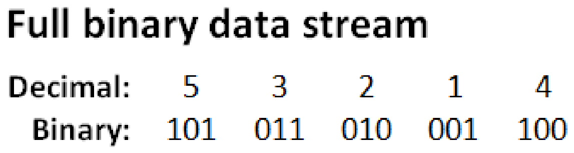
The shifting algorithm has a potential downside with regards to a certain specific type of queries. For queries that cover a large part of the whole dataset, and are therefore affecting many tuples, the shifting algorithm is likely to be much more efficient than the basic algorithm. However for queries that cover only a very small part of the whole dataset there is the potential the shift factor is extremely high and the full left- or right data stream must be (almost) fully scanned before the new median is determined. Tests results support this because this effect is very easy to see when looking at the results for the PEGEL dataset (figures 5.1 and 5.3) when the number of rows involved are between 0 and 1 million. For these results, whereby the queries only cover a maximum of 10% of the whole dataset, the basic algorithm outperforms the shifting algorithm, especially when looking at the group 10 and group 100 test results.

To overcome this problem an improvement has been devised that is able to limit this effect by computing additional structures. This is further discussed later in this chapter in section 5.4. The next section will first discuss a completely alternative algorithm that also works for multi-dimensional datasets.

5.3 The full scanning algorithm

The biggest downside of the shifting algorithm is its unsuitability for multi-dimensional datasets. To solve this problem a completely alternative algorithm called *the full scanning algorithm* has been developed. This algorithm solves the two main problems of the basic algorithm by pre-sorting the full dataset and storing this pre-sorted secondary set as binary data streams thereby avoiding the necessity of fetching each individual tuple from disk. This algorithm works in the following way:

1. Sort a priori the complete data set
2. Store the pre-sorted data set as a binary stream. Figure 5.5 shows the full binary stream for the sample set, discussed in the previous section.
3. To compute a range query, scan the full pre-sorted binary stream and store in-memory all the tuples that match the current query. After the binary stream has been completely scanned the number of tuples in-memory are counted and the middle tuple(s) are return as the median.



Full binary data stream

Decimal:	5	3	2	1	4
Binary:	101	011	010	001	100

Figure 5.5: Pre-computed full data stream for the sample dataset

This algorithm has two advantages when compared to the basic algorithm: (i) the data is pre-sorted thereby avoiding the need to sort in-memory and (ii) scanning the binary stream is significantly faster than scanning the individual tuples from disk. However this algorithm also has a very big potential downside and that is that the full dataset must be scanned to be able to process any query. Additionally this algorithm also needs to store all the tuples that are affected by the query in-memory, although there is an optimization to prevent this, which is further discussed in the next sub-section.

Figure 5.6 shows the test results with the PEGEL data set and the full scanning algorithm. The figure shows a (somewhat) linear relationship between the query performance and the number of affected tuples by a query. Although the scanning time for each query is identical (i.e. the time it takes to scan the full data stream) the decreasing query performance is most likely due to the need to store the affected tuples in-memory. These results show that the shifting algorithm is much better in terms of improving query performance but also in improving the time complexity of the query performance.

The next subsection discusses a variant of the full scanning algorithm that overcomes the need to store all the affected tuples in-memory and provides similar performance improvements as the shifting algorithm.

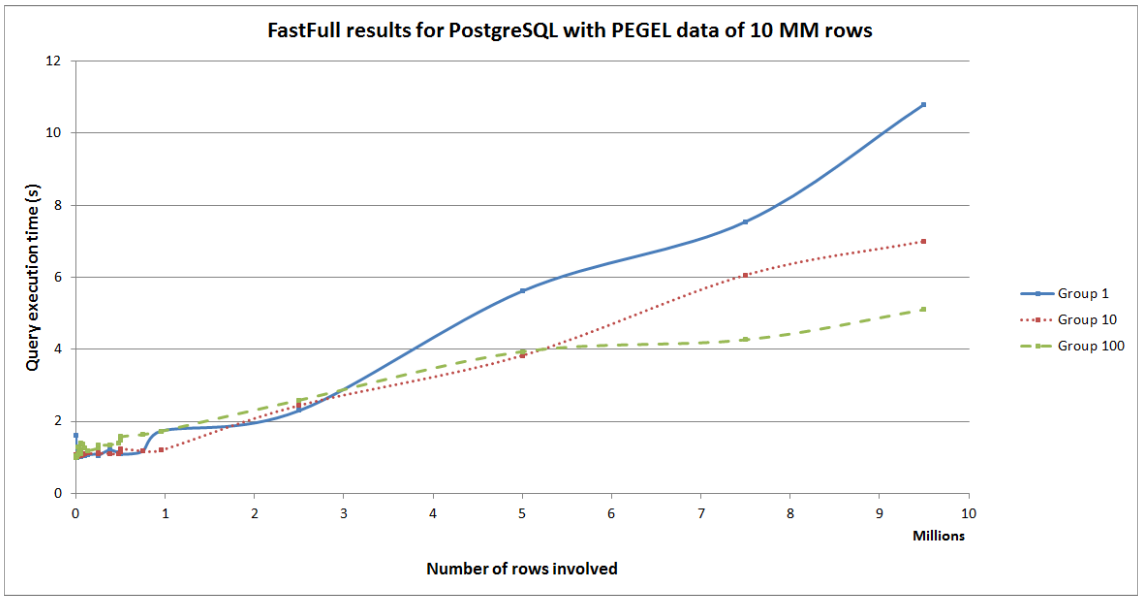


Figure 5.6: Results of the full scanning algorithm and the 10MM PEGEL dataset

5.3.1 Count optimization

The basic variant of the full scanning algorithm must store all the tuples that match the query or current group in-memory to be able to determine the median. However if the total number of tuples affected by the query is known before scanning then no data has to be stored in-memory *and* scanning can be reduced significantly since it can be stopped as soon as the median tuple is scanned. By knowing the exact number of affected tuples the exact position of the median is also known, since this must be exactly the middle tuple(s), i.e. $\frac{count}{2}$.

Generally counting the number of tuples is also a very expensive operation, because this would also require fetching all the tuples from disk but counting is a distributive operation and can be optimized in its own way. This is exactly what has been done already at the University of Twente in the form of the Stairwalker algorithm, as discussed in subsection 2.4.1. Henceforth the counting operation will be considered to be free of cost within the scope this research.

Figure 5.7 shows the test results with the full scanning algorithm and the count optimization for the 10MM version of the PEGEL dataset. This graph shows that the query performance is now almost completely independent of the number of tuples affected by the query and is solely influenced by the time it takes to scan the binary data stream. This becomes even more clear when looking at the results for the 1 MM version of the PEGEL dataset, as seen in figure 5.8. The query performance is almost identical except a factor 10 difference thereby pointing to a linear relationship between the complete size of the dataset and the query performance.

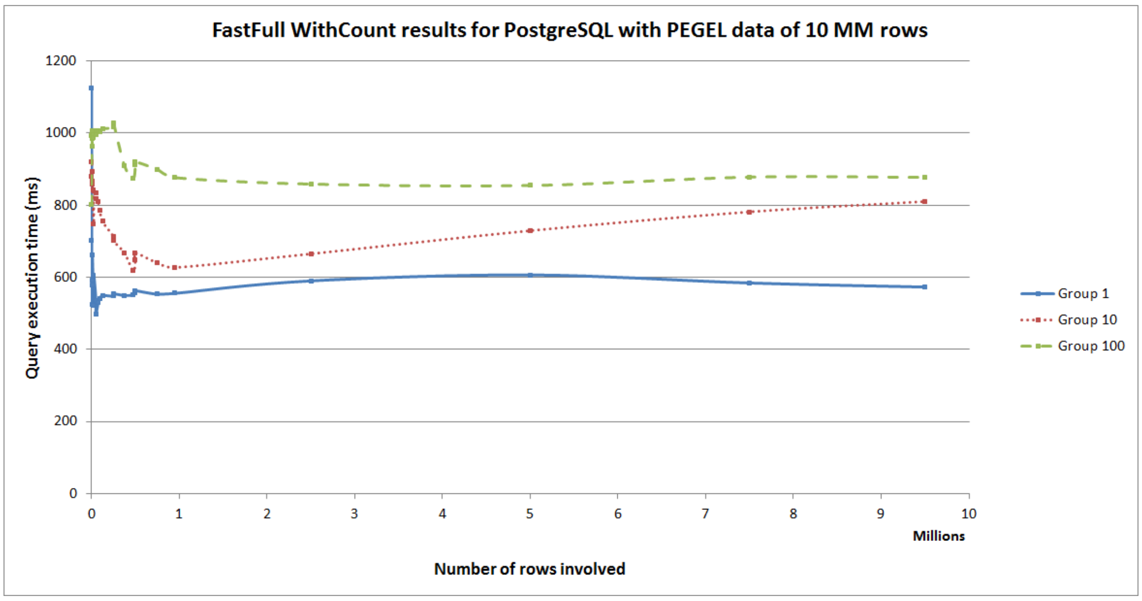


Figure 5.7: Results of the count optimization and the 10MM PEGEL dataset

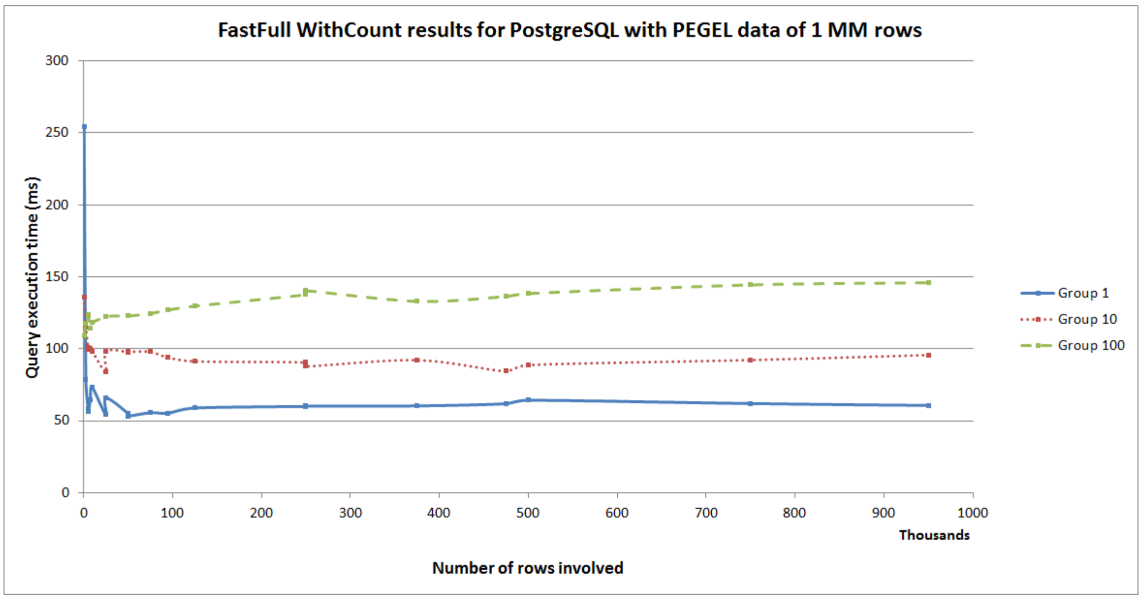
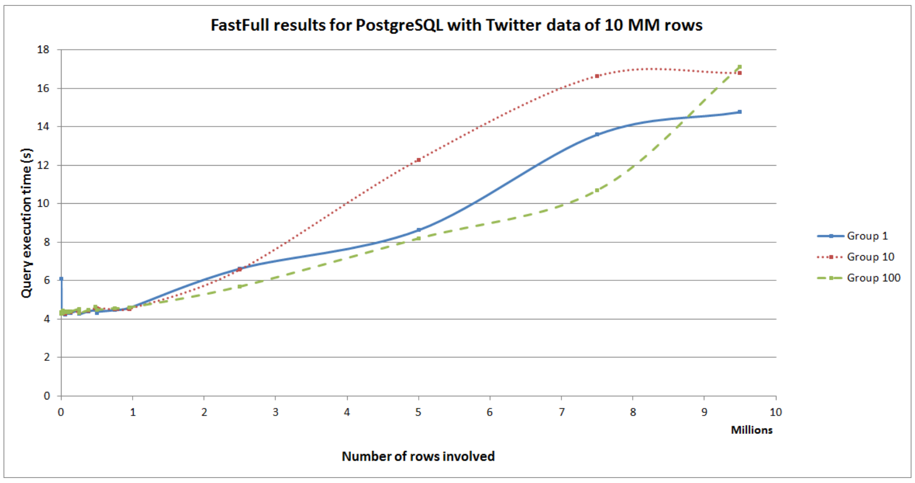
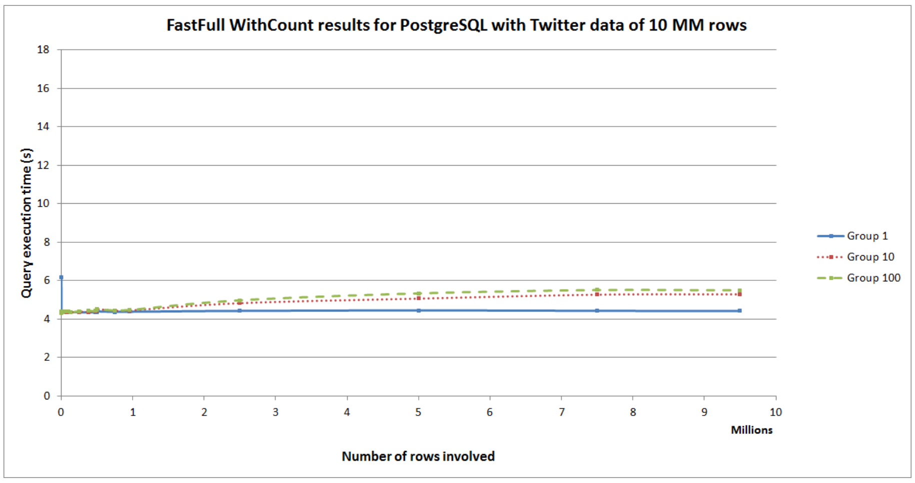


Figure 5.8: Results of the count optimization and the 1MM PEGEL dataset

Considering the full scanning algorithm is primarily meant for datasets with multiple dimensions it is worth looking at the test results of the measurements for this algorithm and the Twitter dataset, which consists of 3 dimensions. Figure 5.9 shows the results of the full scanning algorithm (top graph) and the results of the full scanning algorithm with the count optimization (bottom graph). Both graphs have the same axis. These graph show similar characteristics as the graphs for the single-dimension PEGEL dataset thereby showing that this algorithm is indeed suitable for multi-dimensional datasets, although looking at the absolute query performance it does perform somewhat worse. This is expected though as scanning takes longer due to the fact that there is more data to scan, i.e. each additional dimension must also be included in the full data stream.



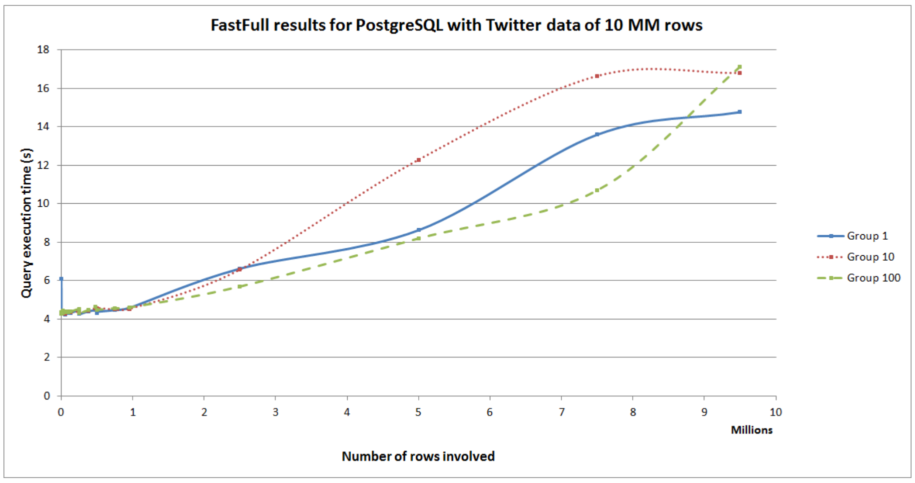
(a) Full scanning algorithm, without count optimization



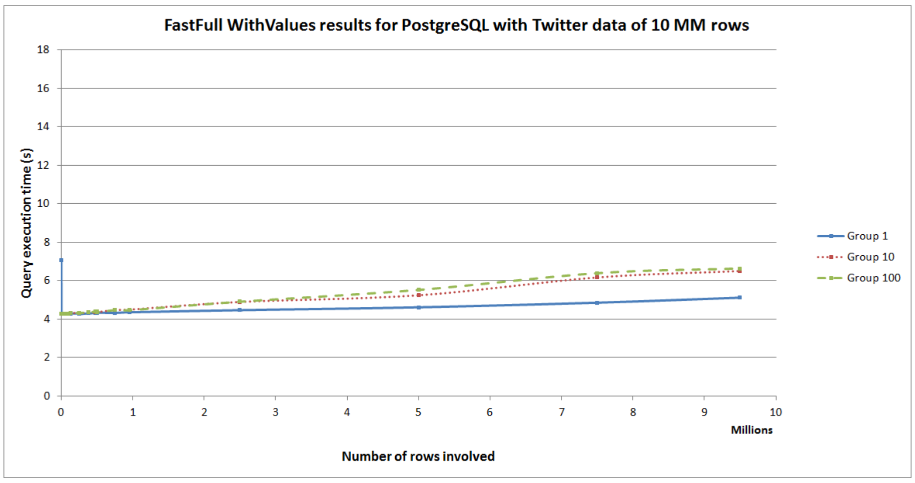
(b) Full scanning algorithm, *with* count optimization

Figure 5.9: Results with the full scan algorithm and the 10MM Twitter dataset

Since the full scanning algorithm also relies on the use of a binary data stream, just like the shifting algorithm, it is also possible to further optimize this algorithm by including the actual tuple values into the data stream instead of only including the tuple key. Figures 5.10 and 5.11 show the results of including the values into the data streams for the basic version of the full scanning algorithm and the variant with the count optimization respectively whereby the top graphs are the results without the values included and the bottom graphs are the results with the values included respectively. These graphs show very clearly that including the actual values in the data stream has the most effect for the full scanning algorithm without the count optimization whereas it has very little effect for the count optimized version. This is primarily due to a property of the Twitter dataset: the tuple keys in this dataset require 8 bytes of storage per key whereas the actual values only require 2 bytes thereby decreasing the amount of memory needed to store all the tuples in memory by the full scanning algorithm. Based on these results and the results whereby the values where included with the shifting algorithm, it is evident that including the values does not have a large influence on the actual disk I/O for a small number of groups.

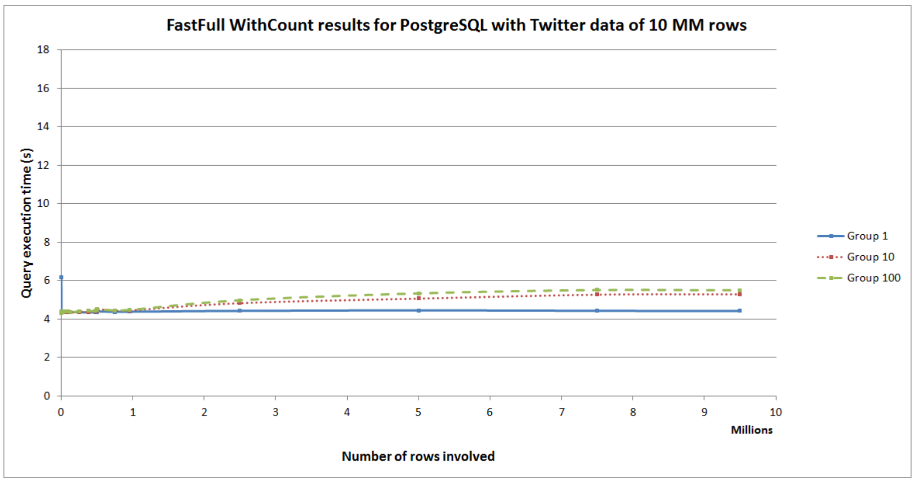


(a) Full scanning algorithm *without* including the values

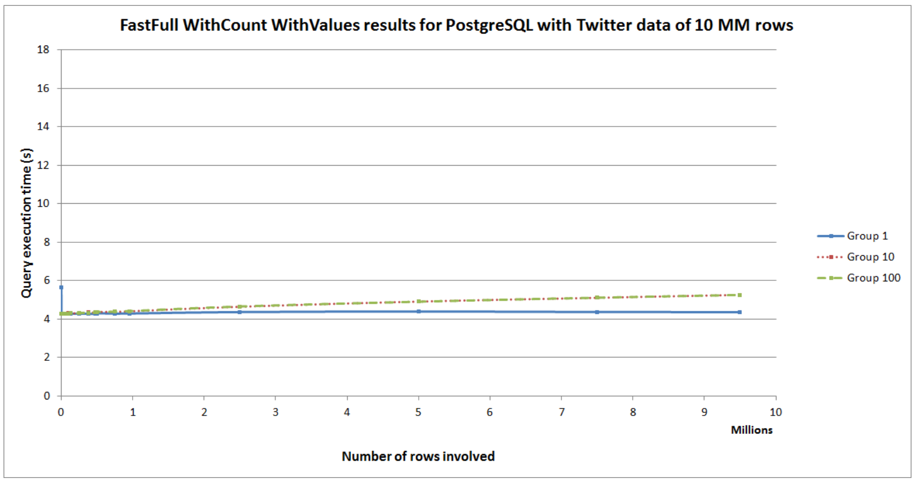


(b) Full scanning algorithm *with* inclusion of the values

Figure 5.10: Results with the full scan algorithm and values and the 10MM Twitter dataset



(a) Full scanning algorithm with count and *without* including the values



(b) Full scanning algorithm with count and *with* inclusion of the values

Figure 5.11: Results with the count optimization and values and the 10MM Twitter dataset.

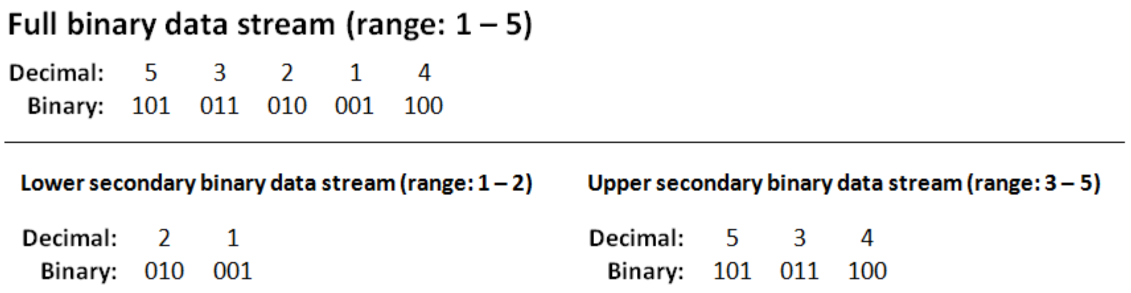
Just like the shifting algorithm, the full scanning algorithm also suffers from the same disadvantage whereby queries or groups that cover a small percentage of the full dataset will likely have worse query performance than the basic algorithm. This problem is likely to be even worse for the full scanning algorithm since the complete data stream must always be scanned or at least until the median has been found (when the count optimization is used). This can be seen in the results when looking at the query performance at the beginning of the graph, which shows that the basic algorithm generally outperforms all the full scanning variants for all datasets when the number of affected tuples is low. The next section discusses a technique that can be used by both the shifting algorithm as well as the scanning algorithm to mitigate this problem and show that this works using test results.

5.4 Sub-groups optimization

This section discusses an optimization technique which is suitable for both the shifting algorithm and the full scanning algorithm as well as all their optimized variants.This optimization technique is called the *sub-groups optimization*. The goal of this technique is to improve the query performance for queries that cover only a fraction of the full data set by pre-computing additional data streams that are specifically tailored towards these smaller queries.

Consider the full scanning algorithm. As explained in the previous section this algorithm depends on the use of a binary data stream that contains the full dataset in a pre-sorted state. To determine the median of a query, even a query that for example only covers 2 individual tuples, will require the scanning of the full data stream to be able to ensure that all tuples affected by the query are found. With the sub-groups optimization two extra data streams would be pre-computed whereby the first extra data stream contains the left half of the full dataset in a pre-sorted state and the second extra data stream contains the right half of the full dataset. With these two extra data streams determining the median of smaller queries can now be done with either extra data stream, thereby reducing the time for scanning in half.

Figure 5.12 demonstrates the use of this optimization technique for the sample dataset. The full binary data stream has already been shown in the previous section. The additional "lower" and "upper" secondary data streams are pre-computed as part of the sub-groups optimization. A median query with time range of for example [1,2] or [3,4] can now be computed with the use of the lower- and upper secondary binary data streams respectively thereby lowering the scanning time.



Full binary data stream (range: 1 – 5)

Decimal:	5	3	2	1	4
Binary:	101	011	010	001	100

Lower secondary binary data stream (range: 1 – 2)

Decimal:	2	1
Binary:	010	001

Upper secondary binary data stream (range: 3 – 5)

Decimal:	5	3	4
Binary:	101	011	100

Figure 5.12: Pre-computed sub-group data stream for the sample dataset

This optimization technique also works for the shifting algorithm in a similar way, whereby an additional left- and right data stream is generated for each subgroup. Figure 5.13 shows test results for the sub-groups optimization with the shifting algorithm and the 10MM version of the PEGEL dataset. The left part of the figure are the results of the shifting algorithm without the optimization and the part of the figure are the results of the algorithm with the sub-groups optimization. The effect of the optimization is most clearly visible for the top line (which are the results of the group-100 queries) whereby the query performance for the smallest queries, whereby the smallest number of tuples are affected, is slightly improved by the optimization. The effect is even more clearly noticeable when looking at the query performance with the full scanning algorithm and the KNMI dataset, as seen in figure 5.14 and even more so when looking at the results with the full scanning algorithm with the count and values optimizations for the Twitter dataset, as seen in figure 5.15.

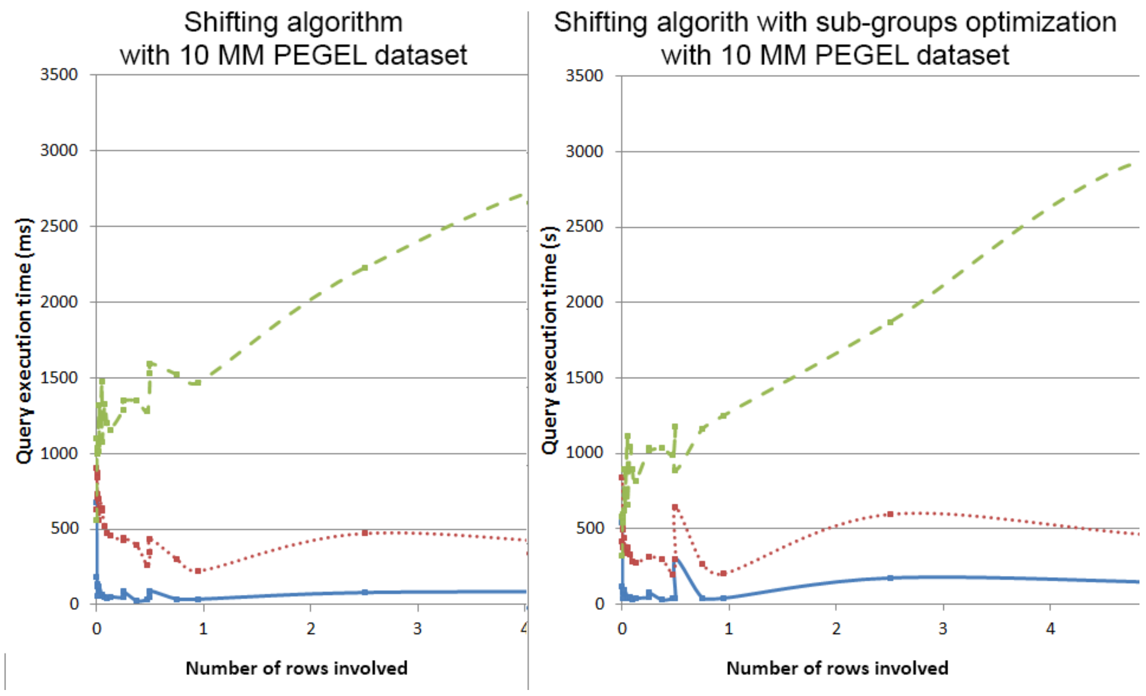


Figure 5.13: Comparison of the shifting algorithm and the sub-groups optimization

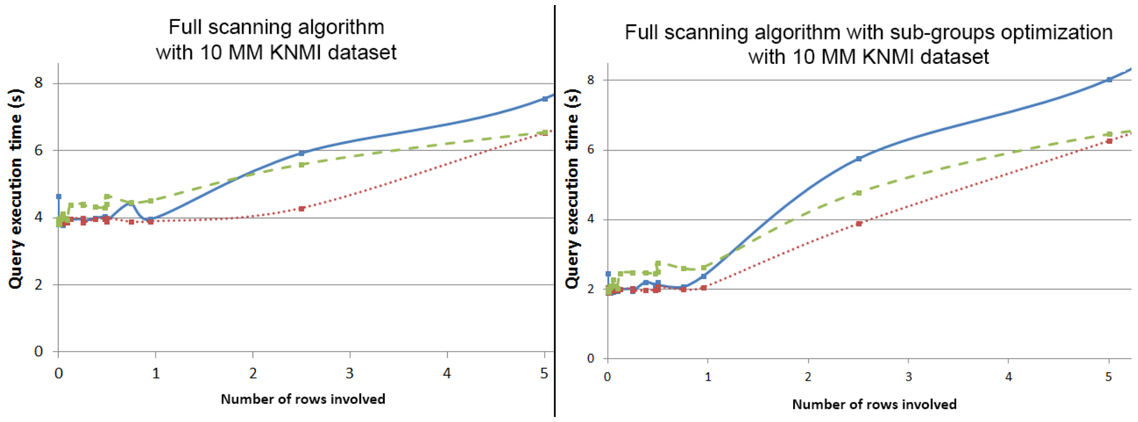


Figure 5.14: Results of the full scanning algorithm and the sub-groups optimization

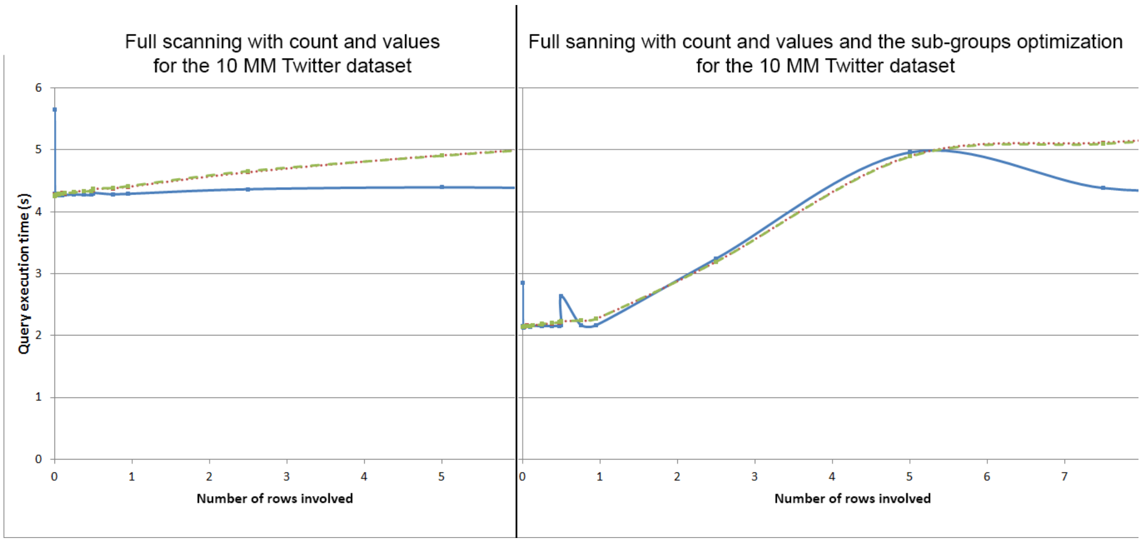


Figure 5.15: Results of the full scanning algorithm with count, values and sub-groups optimization

The downside of this optimization technique is that it requires more storage and more (pre-)processing time. For each additional sub-group level the whole dataset must be pre-sorted and stored as additional data streams. Each additional level of sub-groups is also only able to improve the query performance for a (small) subset of queries but comes at a cost therefore a balance between the number of levels and achievable query performance is necessary. In some cases it may be more advantageous to use the basic algorithm for very small queries and switch to an alternative algorithm for queries that cover a bigger range. In the next chapter the storage costs of the algorithms will be discussed in further detail.

The next section will provide a conclusion of the analysis into the development of alternative algorithms for determining the median.

5.5 Conclusion

This chapter has outlined and analyzed the development into finding new algorithms and various optimization variations for improving query performance for median (aggregation) queries. The starting point of this development was the basic algorithm, which is at the moment the standard algorithm for most database engines. Based on the test results and investigation into this algorithm two main problems have been identified with this algorithm that have a large negative effect on query performance: (i) all tuples must be fetched from disk and (ii) to determine the median all tuples must be stored and sorted in-memory.

To solve these problems an alternative algorithm has been developed called the shifting algorithm. This algorithm no longer requires fetching all tuples from disk and storing and sorting them in-memory thereby solving both main problems of the basic algorithm. Experiments with the test datasets and test queries have indeed shown that this algorithm is able to greatly improve the query performance and is able to deliver a near constant query performance independent of the number of tuples affected by the query. This algorithm does require the use of two pre-computed binary data streams and for each tuple extra information has to be pre-calculated before this algorithm is able to work as designed.

The major downside of the shifting algorithm is its unsuitability for datasets with multiple dimensions. During this research tests have been done it with a 2-dimensional dataset (the KNMI dataset) however this dataset has a very limited secondary dimension with only 10 distinct values. As the number of dimensions increase, and the distinct values per dimension, the computational complexity of the shifting algorithm increases very quickly thereby making it unsuitable.

That is why another algorithm, called the full scanning algorithm, has been developed, which is more suitable for higher dimensional datasets. This algorithm depends on a single pre-computed binary data stream, whereby the data is pre-sorted and stored so that this does not need to happen at query-time. To be able to compute the results of a median query the full data stream is scanned to find all the tuples that match the query and then the middle tuple is returned as median since all the tuples are already pre-sorted.

The basic variant of this algorithm has to store all the tuples in-memory to be able to determine the median. To solve this problem a variant of the full scanning algorithm has been developed whereby the number of affected tuples is determined before actually scanning the binary data streams which prevents the need to store all the tuples in-memory and reduces the scanning time because scanning can be stopped as soon as the median tuple is found. This variant depends on being able to determine the number of affected tuples (i.e. a count) in a very efficient way, for example with the Stairwalker algorithm as previously discussed in sunsection 2.4.1. In test results this time has not been taken into account.

An optimization technique for both the shifting and the full scanning algorithm has also been created whereby the actual measure values is included into the binary data streams instead of only the key of each tuple with the goal of reducing disk IO once the correct tuple has been identified as the median. Test results have shown that this optimization has s mixed success rate and is primarily able to improve the query performance if the storage size of the measure value is smaller (i.e. 2 bytes) than the tuple key (i.e. 8 bytes). Test results also show that this optimization might reduce disk IO more significantly if a query is grouped into many groups (i.e. number of groups $>>$100).

Finally an optimization has been developed which is used to improve the query performance for queries that cover a small part of the full dataset, which generally have worse query performance with the alternative algorithms than the basic algorithm. This optimization technique, called the sub-groups optimization, relies on splitting the full dataset into multiple parts (e.g. halves, fourths, tenths, etc) and pre-computing the data streams independently for each part of the dataset. If and when a small query fits into one of the parts then this part is used to actually calculate the median instead of the data stream for the full dataset. This generally results in improved query performance because the time to scan the data stream is reduced due to the smaller size of the stream. Test results have shown that this optimization is successful in improving the query performance for smaller queries.

The next chapter will evaluate this research in a structured way by analyzing, for each algorithm and each optimization, the improvement in query performance and the storage requirements. This analysis will be based on the test results, which have been partly discussed already in this chapter.

Chapter 6

Evaluation

This chapter will evaluate the developed algorithms and optimizations by systematically analyzing them on two different criteria: query performance and storage requirements. The first section will analyze the query performance of each algorithm and optimization and the second section will analyze the storage requirements. In the third section the findings of the evaluation will be summarized. Finally the last section of this chapter will provide a conclusion of the evaluation of each algorithm and optimization.

6.1 Query performance

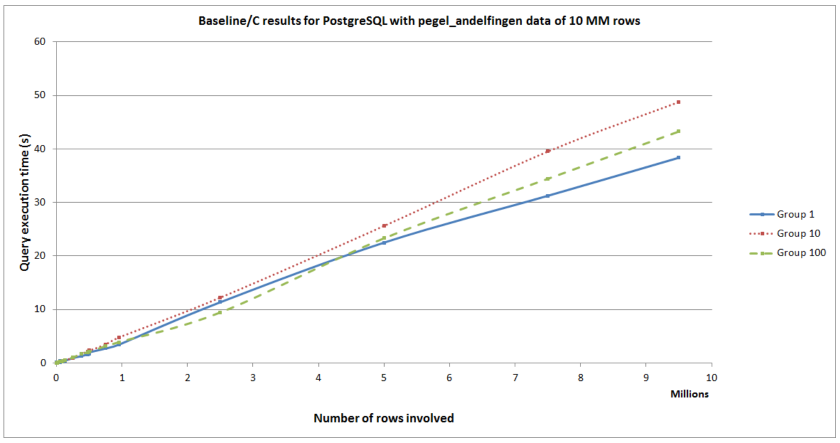
This section will discuss the query performance of each algorithm and optimization, whereby there is a focus on the time complexity of the query performance and the resulting scalability of this time complexity. In the following subsections each algorithm, starting with the basic algorithm, and optimization as outlined in the previous chapter will be analyzed.

All the test results in the following subsections have been collected using the experimental setup described in chapter 4. This setup consists of three test data sets (the PEGEL dataset, the KNMI dataset and the Twitter dataset), a set of randomly generated test queries for each data set, a fixed test platform to run experiments on and a Java tool to automatically run all experiments in a standardized way. Each test query has been executed with three different group-by queries: as a single group, divided into 10 groups and divided into 100 groups. Full details on this setup can be found in chapter 4.

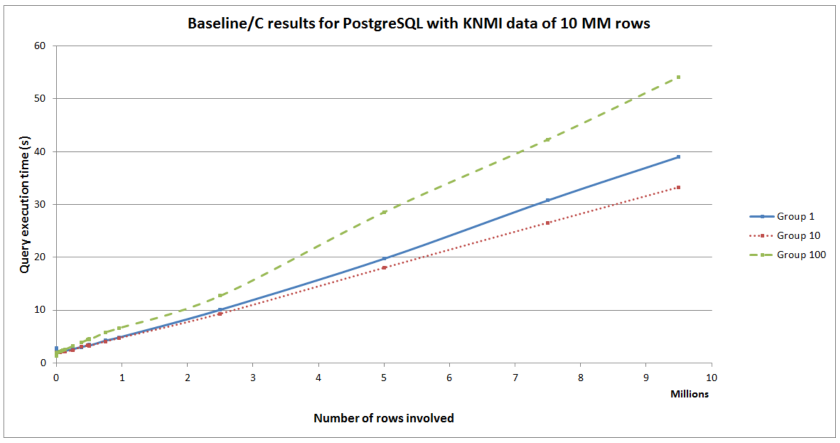
6.1.1 Basic algorithm

The query performance of the basic algorithm serves as the baseline to which the performance of the alternative algorithms are compared to. Figure 6.1 shows the query performance of the basic algorithm for all three test datasets. All three graphs show a similar relationship between the number of affected tuples affected by the query (the x-axis of the graphs) and the query performance (indicated by the query execution time on the y-axis). This is a linear relationship: as the number of affected tuples increase the query execution time increases at a similar rate (and therefore the query performance decreases). It is important to note that the absolute query performance is almost the same for each test dataset despite the difference in data distribution, number of dimensions and measure values.

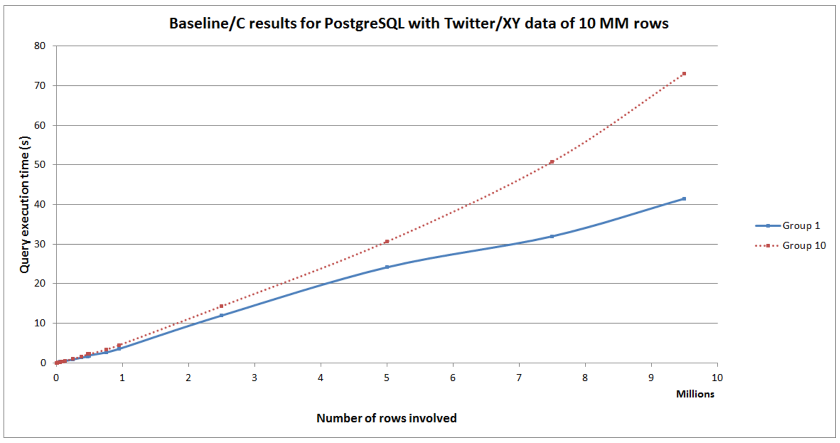
The test results, as shown in figure 6.1, show that the grouping parameter of the aggregation queries also has an influence on the query performance although this is a very unclear effect because in one case the group-1 queries have the best query performance (e.g. for the PEGEL dataset) and in another case the group-10 queries have the best performance (e.g. for the KNMI dataset). It is unclear sure why this is, but one possible reason may be overhead of the internal grouping mechanism of the database engine. For the Twitter dataset the group-100 queries had such bad performance that reliable test results could not be obtained.



(a) PEGEL results



(b) KNMI results



(c) Twitter results

Figure 6.1: Test results for the basic algorithm

Having analyzed the query performance of the basic algorithm it is now possible to analyze the performance of the alternative algorithms. The next subsection will evaluate the shifting algorithm.

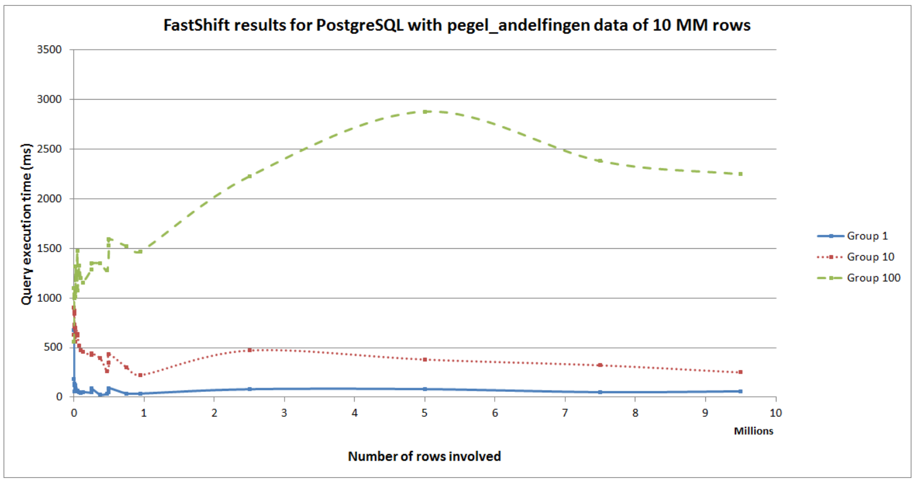
6.1.2 Shifting algorithm

The test results of the shifting algorithm, as shown in figure 6.2 show that the relationship between the number of tuples involved in a median (aggregate) query and the query performance no longer exists; the affected number of tuples no longer has any effect on the query performance. This is shown in figure 6.2 as the graphs are almost completely flat, especially when looking at the results of the group-1 and group-10 queries. The test results of the group-100 queries has an increase at around 5 MM rows, which is an indication that at this point the actual shifting performed by the algorithm is at its highest. This is most likely due to the actual data distribution of both data sets.

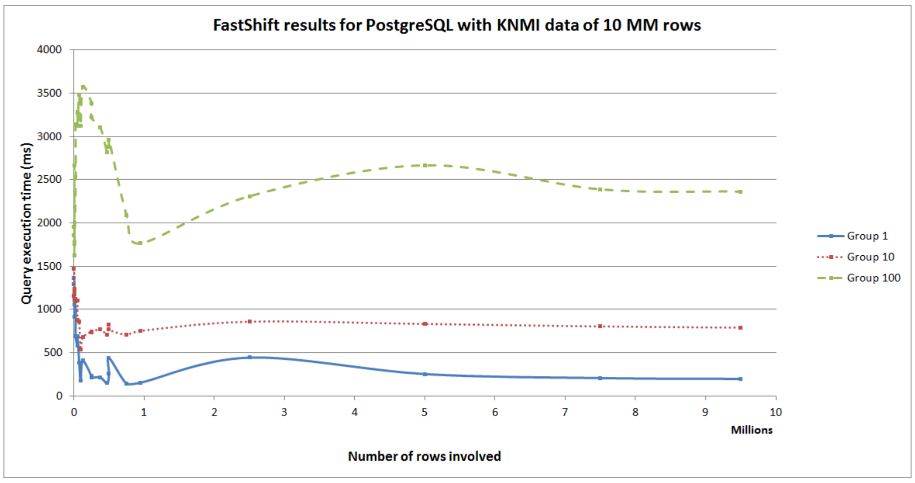
It is interesting to note the peaks at the beginning of both graphs. Clearly very small queries, covering only a tiny part of the total data set, have the worst query performance for both datasets and all grouping configurations. Most likely, this is due to the fact that for these queries the shift factor may be quite small but the actual scanning of the binary streams takes much longer because the involved rows are spread over the full binary streams.

Although the query performance is no longer linear related to the number of involved rows it does not mean it is completely fixed. Instead, there is now a linear relationship between the size of the full dataset and the query performance. This can be seen when looking at figure 6.3, which shows the the test results with the 1 MM version of the PEGEL dataset (top graph) and the 10 MM version of the PEGEL dataset (bottom graph). The query performance of the 10 MM version is approximately 2x as slow as the query performance of the 1 MM version.

The test results also show a linear relationship between the number of groups of a query (e.g. group into 1, 10 or 100 groups) and the query performance. For example when looking at figure 6.2b, the test results for the group-100 queries are approximately 5x as slow as the tests results for the group-10 queries.

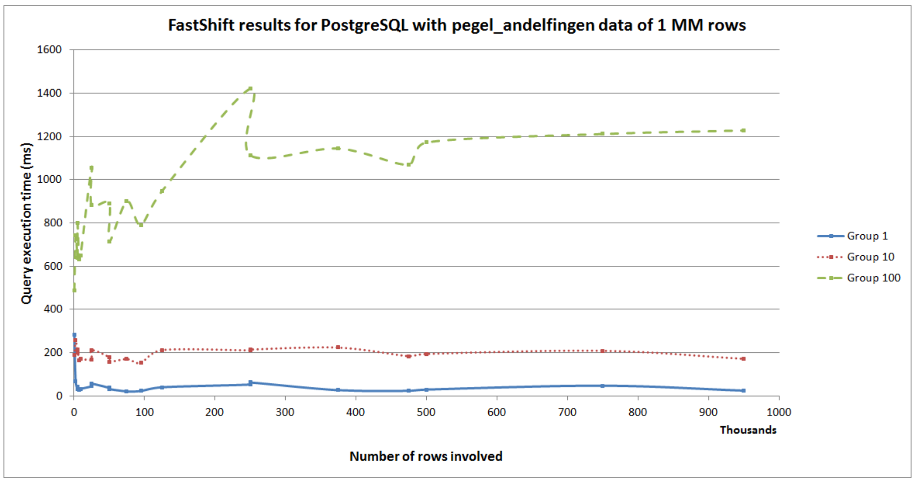


(a) PEGEL results

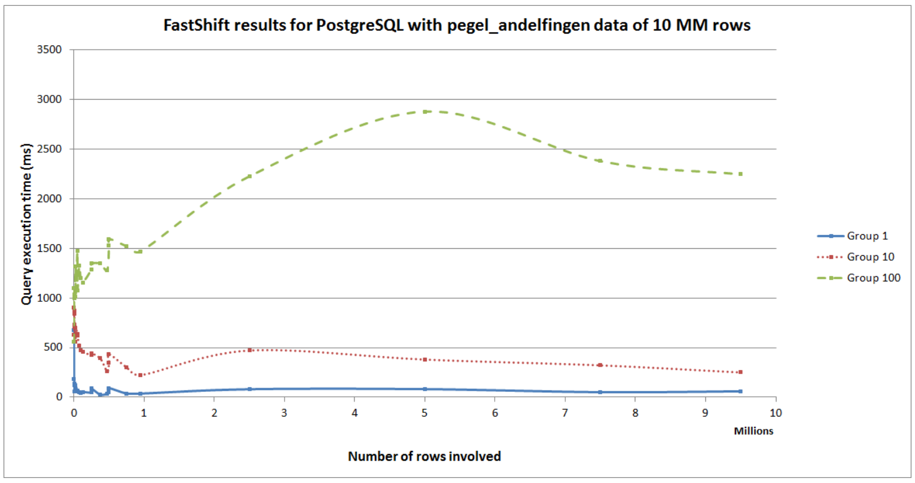


(b) KNMI results

Figure 6.2: Test results for the shifting algorithm



(a) PEGEL 1 MM results



(b) PEGEL 10 MM results

Figure 6.3: Test results for the shifting algorithm for the PEGEL dataset of various sizes

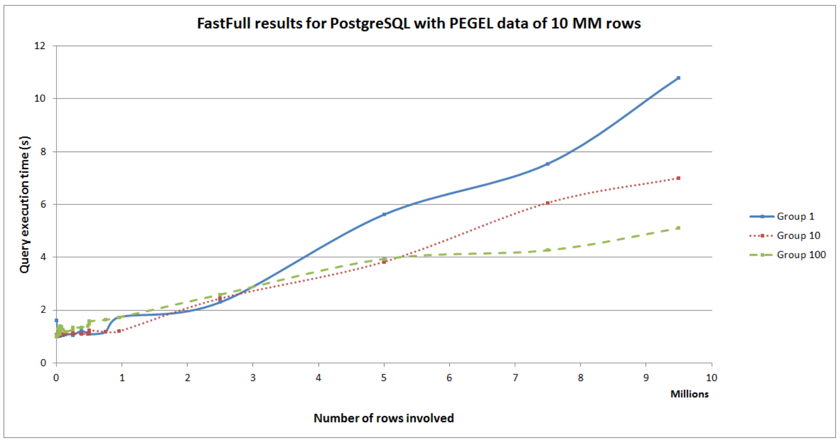
The next subsection will analyze the query performance of the full scanning algorithm and how it compares to the baseline performance.

6.1.3 Full scanning algorithm

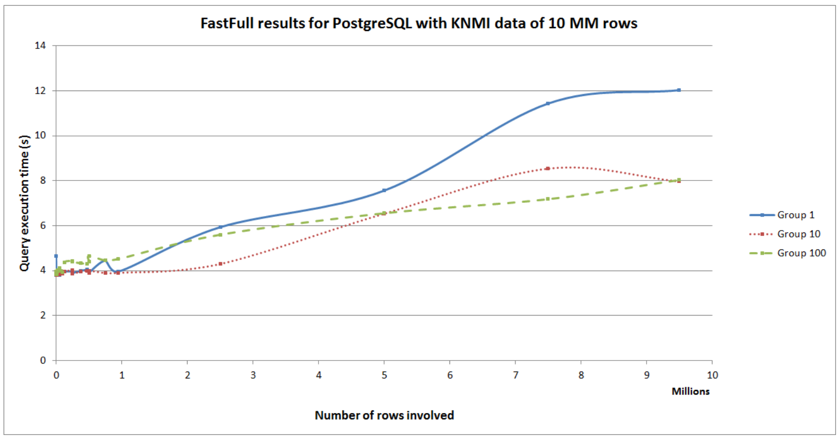
The test results of the query performance of the full scanning algorithm are shown in figure 6.4. These test results show for all test datasets a very similar performance profile as the test results of the basic algorithm, i.e. a linear relationship between the number of tuples and the query performance although absolute query performance of the full scanning algorithm is approximately 10x better than the performance of the basic algorithm.

Since these results show that the full scanning algorithm still has the same performance behavior as the basic algorithm it is less likely to scale as well as the shifting algorithm. As already mentioned in the previous chapter, this is likely due to the fact that the full scanning algorithm needs to (temporarily) store all affected tuples in memory before being able to determine the median. Clearly this is a costly operation and should be avoided.

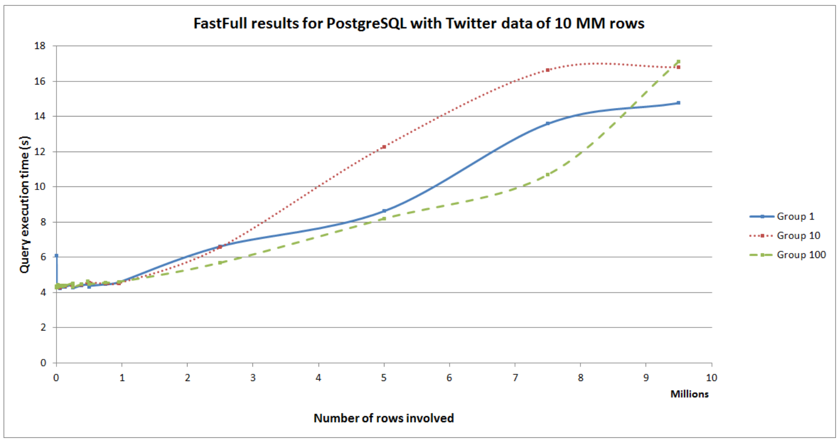
Unlike the shifting algorithm, with the full scanning algorithm the grouping configuration has a positive effect on the query performance, i.e. as the number of groups increases the query performance also improves. This is exactly as expected, because only the tuples for the each group needs to be stored in memory which means that for the group-1 queries all the tuples involved with the query need to be stored in memory whereas for the group-100 queries only the tuples for a single group need to be stored in memory. The test results in figures 6.4a and 6.4b confirm that this is indeed the case although the results in figure 6.4c do not fully support this.



(a) PEGEL results



(b) KNMI results



(c) Twitter results

Figure 6.4: Test results for the full scanning algorithm

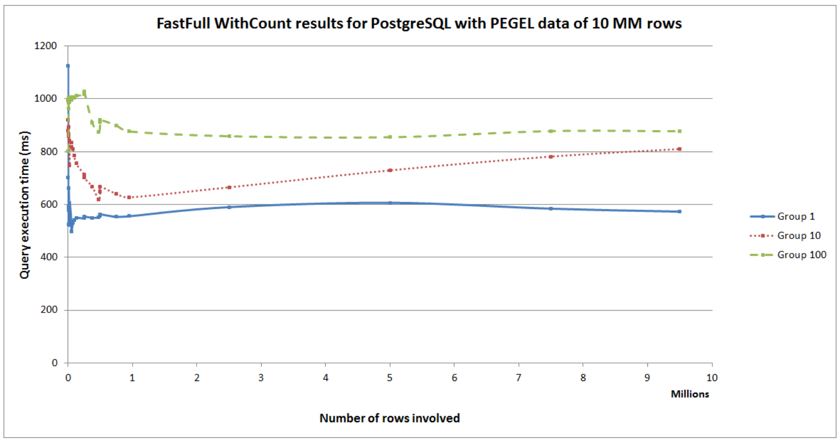
The next subsection will evaluate the query performance of the variant of the full scanning algorithm whereby the tuple count is known beforehand thereby avoiding the requirement to store all the tuples in memory.

6.1.4 Full scanning with count variant

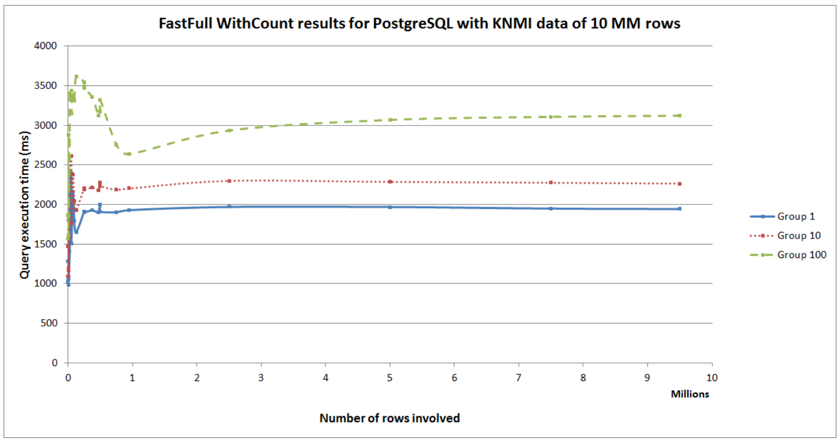
Figure 6.5 shows the test results of the query performance measurements with the count variant of the full scanning algorithm. These graphs show that, similar as with the shifting algorithm, the linear relationship between the query performance and the number of tuples affected by the query has disappeared. The query performance is no longer affected directly by the number of tuples but instead it is now determined by the total size of the dataset, as clearly shown by figure 6.6. This figure shows the results of the full scanning with count algorithm with the 1 MM version (top graph) and 10 MM version of the PEGEL data set (bottom graph). Although not very clearly visible in the graphs, this relationship is an linear relationship, i.e. as the size of the dataset increases the query performance decreases in a linear fashion. The shifting algorithm also has a linear relationship between the query performance and the size of the full dataset but it's linear growth is more gradual.

It's interesting to note that the grouping configuration has the opposite effect on the query performance with this variant when compared to effect it has with the basic variant of the full scanning algorithm. Now the query performance degrades as the number of groups increases, as clearly shown in all three graphs in figure 6.5 by comparing the query performance of the group-100, group-10 and group-1 queries.

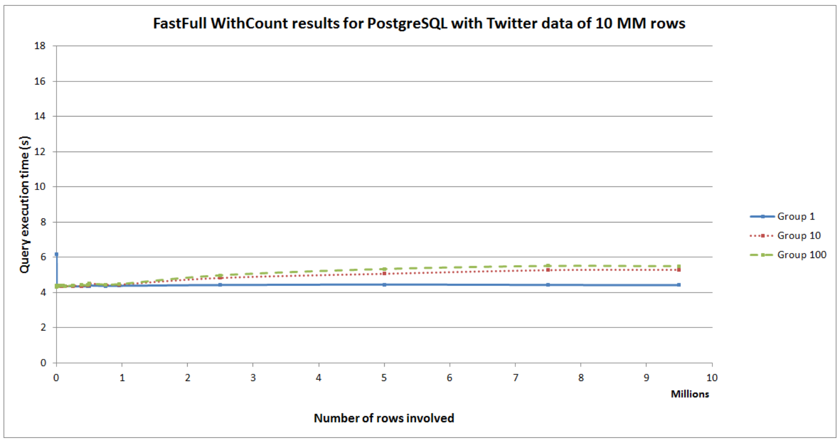
All three graphs also show high deviations at the beginning, which involves smaller queries (i.e. number of involved rows less than 2 MM). This is due to the fact that the tuples involved with these smaller queries are likely to be spread all over the binary stream, which means there is a high chance that a very large part of the binary stream must be scanned before the median tuple is found and scanning can be aborted. This is exactly the use-case for which the sub-groups optimization has been developed, which is evaluated later on in this chapter in sub-section 6.1.6.



(a) PEGEL results

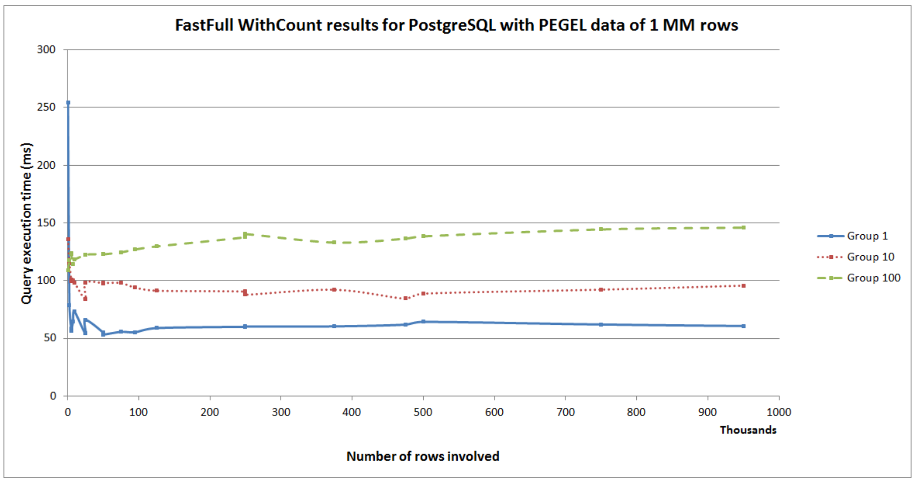


(b) KNMI results

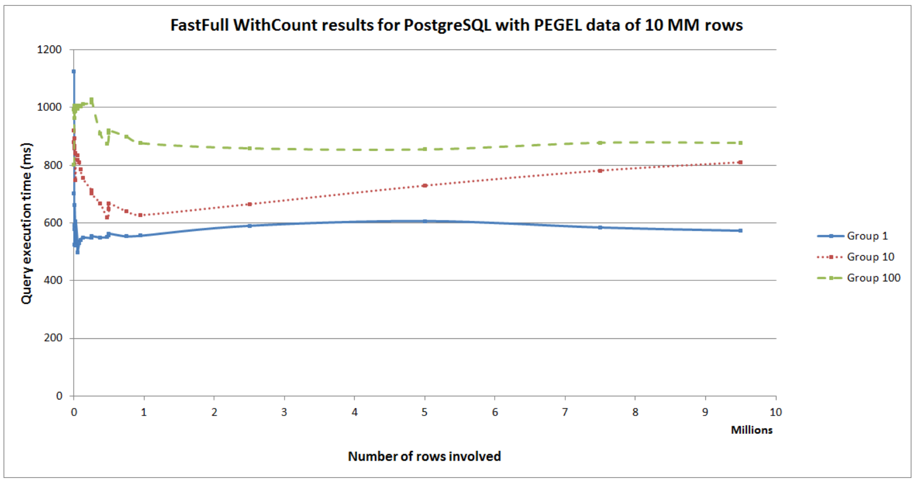


(c) Twitter results

Figure 6.5: Test results for the full scanning with count variant



(a) PEGEL 1 MM results



(b) PEGEL 10 MM results

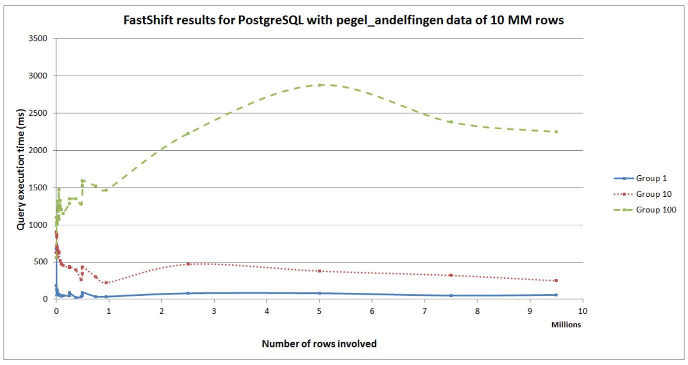
Figure 6.6: Test results showing the relationship between query performance and the dataset size

The next subsection will evaluate the effect of optimizing the shifting algorithm and the full scanning algorithm whereby the actual measure values are included into the data streams instead of the tuple keys.

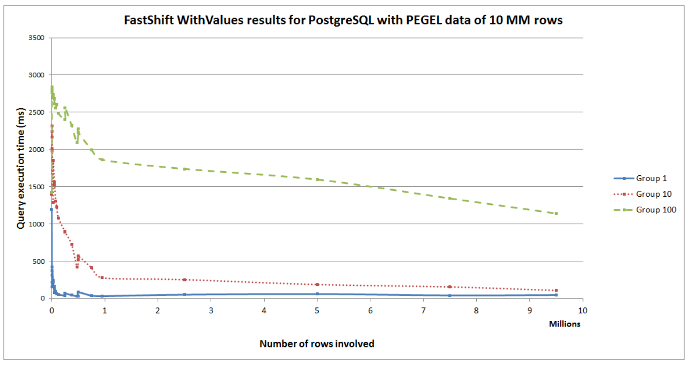
6.1.5 Including values optimization

Figure 6.7 shows the test results of the query performance for the shifting algorithm without and with the values included. These results show that this optimization has a mixed effect on the query performance and on the time complexity of the performance. For the PEGEL data set the query performance now actually improves as the number of involved tuples increases although this effect is primarily seen with the group-100 and group-10 queries. This is an indication that the use of this optimization does have some effect in reducing the additional index lookups for fetching the actual measure values. However, it does not have this effect on the KNMI data set although the query performance for the KNMI data set is improved by approximately 25%. Therefore it is difficult to definitively conclude that this optimization has the desired effect as theorized in section 5.2.

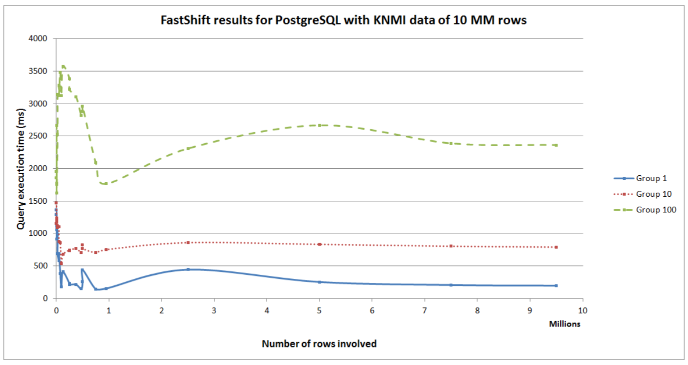
It is important to note that in some cases the query performance is actually degraded, for instance when looking at the beginning of figure 6.7b, which shows that the query performance of the smaller queries is worse than the query performance of the same queries when the values are not included into the binary streams. This is due to the fact that these queries tend to require much more scanning of the streams and this takes longer when the values are included, especially for the PEGEL dataset since this dataset does not require the inclusion of an additional artificial tuple key. This becomes even more evident when looking at the result for the KNMI dataset, which does not suffer from this problem since scanning the streams actually becomes faster for this dataset since the artificial tuple keys are replaced by the (smaller) measure values in the binary streams.



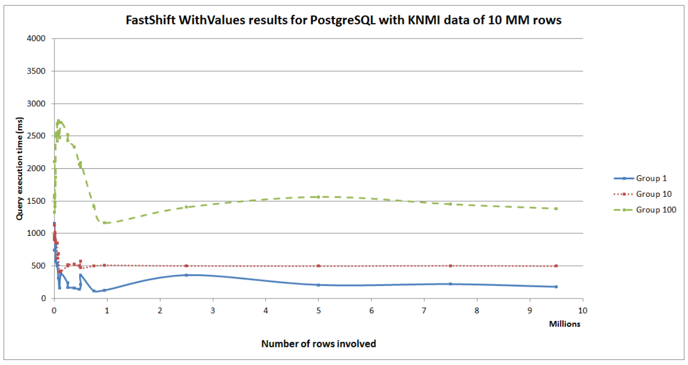
(a) PEGEL results without values



(b) PEGEL results with values



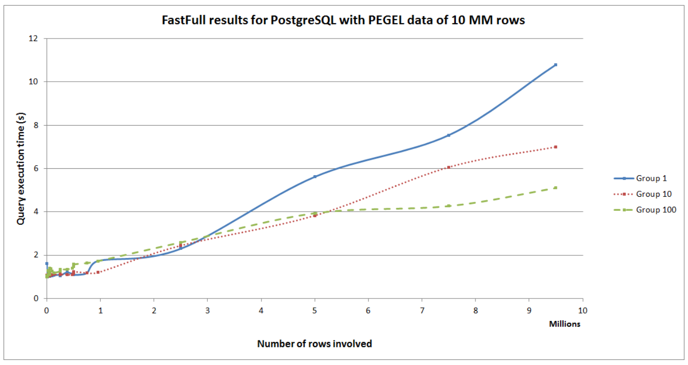
(c) KNMI results without values



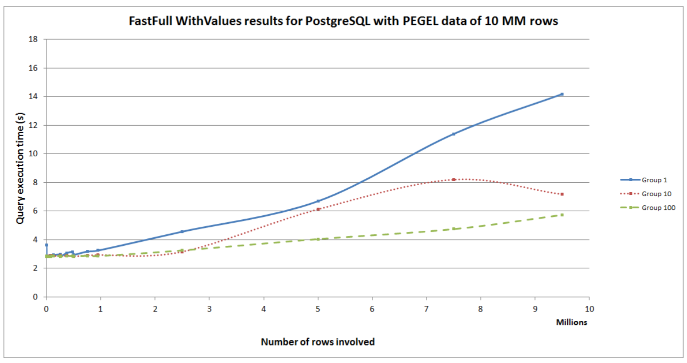
(d) KNMI results with values

Figure 6.7: Test results for the with values optimization for the shifting algorithm

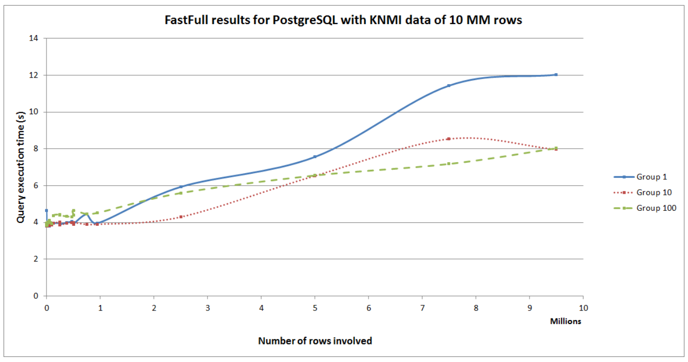
This optimization can also be used with the full scanning algorithm. Test results of this are shown in figure 6.8. These results show that the impact of including the values into the binary streams can either worsen query performance (e.g. for the PEGEL dataset, as seen in figures 6.8a - 6.8b) or improve query performance by more than 50% (e.g. for the Twitter dataset, as seen in figure 6.8e - 6.8f). This is primarily dependent on whether the actual measure value takes up less storage than the artificial key or not which in turns leads to decreased or increased scanning time.



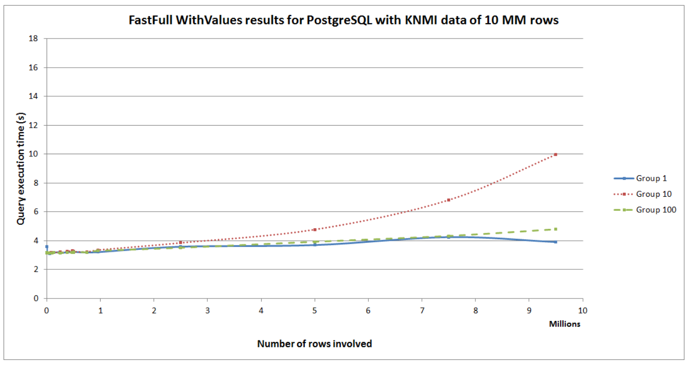
(a) PEGEL results without values



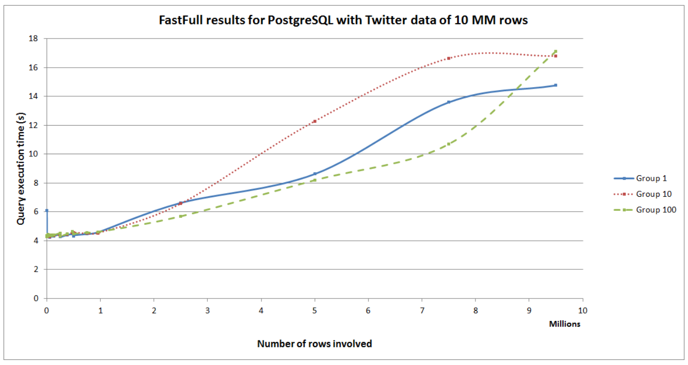
(b) PEGEL results with values



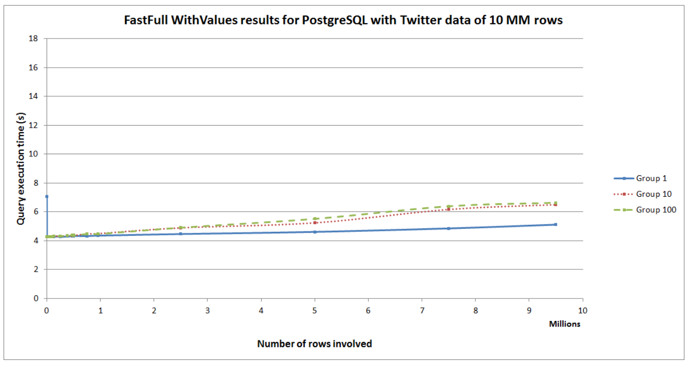
(c) KNMI results without values



(d) KNMI results with values



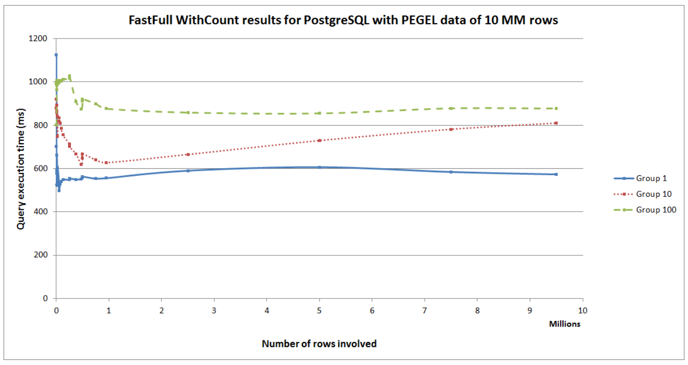
(e) Twitter results without values



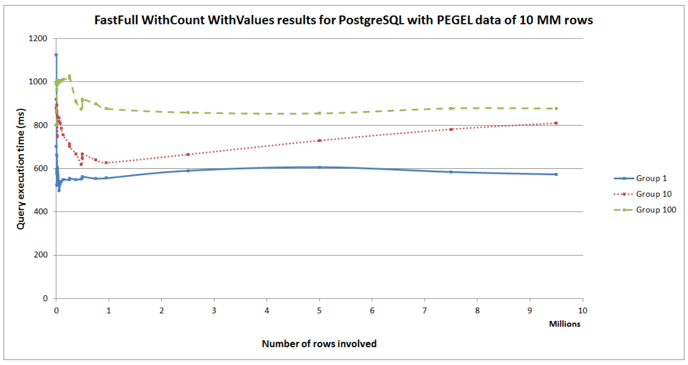
(f) Twitter results with values

Figure 6.8: Test results for the with values optimization for the full scanning algorithm

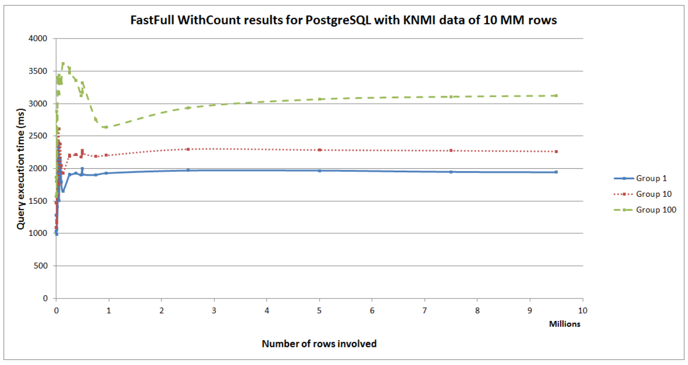
This optimization can also be used with the count variant of the full scanning algorithm. Test results of this are shown in figure 6.9. These results unfortunately show that including the values has very little effect on the time complexity of the query performance or the query performance for the full scanning with count algorithm.



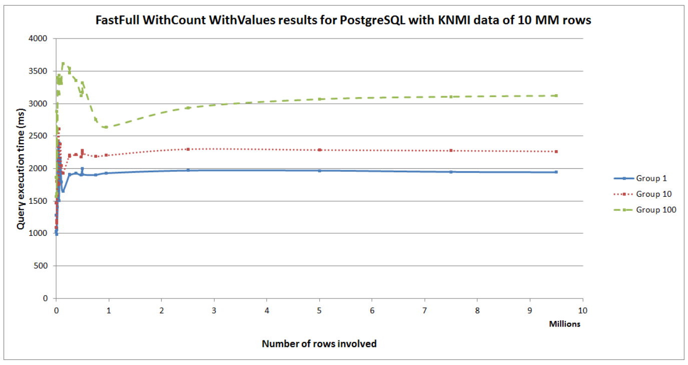
(a) PEGEL results without values



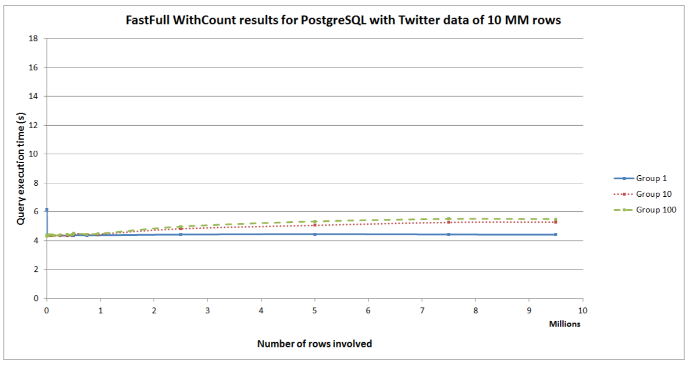
(b) PEGEL results with values



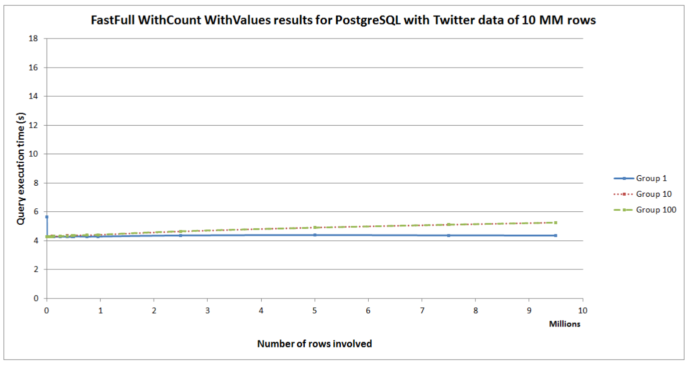
(c) KNMI results without values



(d) KNMI results with values



(e) KNMI results without values



(f) KNMI results with values

Figure 6.9: Test results for the with values optimization for the full scanning with count algorithm

The next subsection will analyze the effects on the query performance of the sub-groups optimization.

6.1.6 Sub-groups optimization

This subsection will evaluate the effects of the sub-groups optimization on the query performance for the shifting algorithm and the count variant of the full scanning algorithm. Figure 6.10 shows the test results of the query performance of the shifting algorithm without the sub-groups optimization (left graph) and with the sub-groups optimization (right graph) for the PEGEL dataset. Due to technical issues test results of this optimization with the shifting algorithm for the KNMI dataset are not available.

The effect on the query performance is fairly minor and can only really be seen when looking at the results of the top line (these are the results of the group-100 queries) whereby the beginning shows improved (absolute) query performance. Clearly, for the shifting algorithm this does not change the time complexity of the algorithm at all. Oddly enough the test results of the right graph show a new peak for the group-1 and group-10 queries at around 0.5 MM rows involved, which is not present in the left graph. The exact cause of this has not been systematically investigated but based on the mechanism behind the shifting algorithms works it is likely that this specific deviation occurs because of an increase in the shifting factor and/or scanning time due to the location of the median of the sub-group versus the median of the full data set.

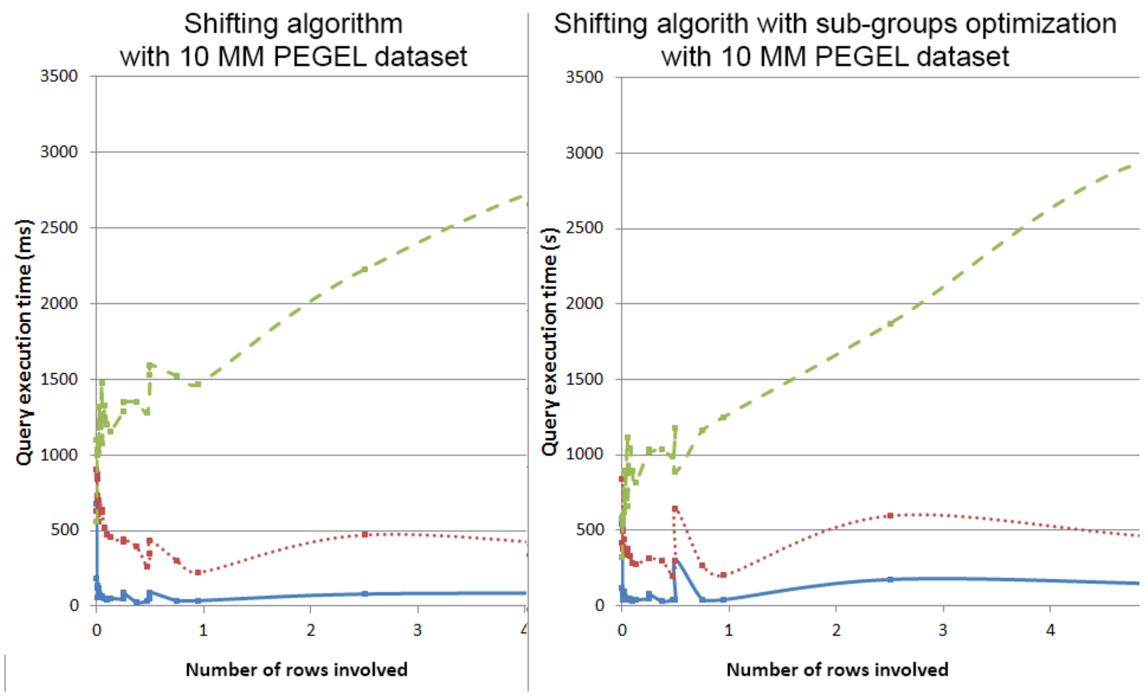


Figure 6.10: Test results for the sub-groups optimization with the shifting algorithm

The test results for the sub-groups optimization in conjunction with the fullscan algorithm and its count variant show more promise. Figure 6.11 shows the test results of the sub-groups optimization with the basic variant of the full scanning algorithm and the KNMI dataset. These test results clearly show that the optimization has a positive effect on the query performance of smaller queries (i.e. whereby the number of rows is less than 5 MM).

This is also very clearly seen when looking at the test results of the sub-groups optimization with the count variant of the full scanning algorithm, as seen in figure 6.12. Again these results show that this optimization is able to significantly improve the query performance of smaller queries, although the effect is clearly no longer visible when the number of involved rows is more than 5 MM. Test results of this optimization in conjunction with the optimization whereby the values are included, as shown in figure 6.13 show a similar improvement in query performance.

Test results of the sub-groups optimization has clearly shown that this optimization is able to significantly improve query performance of smaller queries and is able to reduce high deviations in query performance for such queries. However the effect of this optimization is strictly limited to smaller queries, although the exact definition of a ”smaller” query is not entirely fixed.

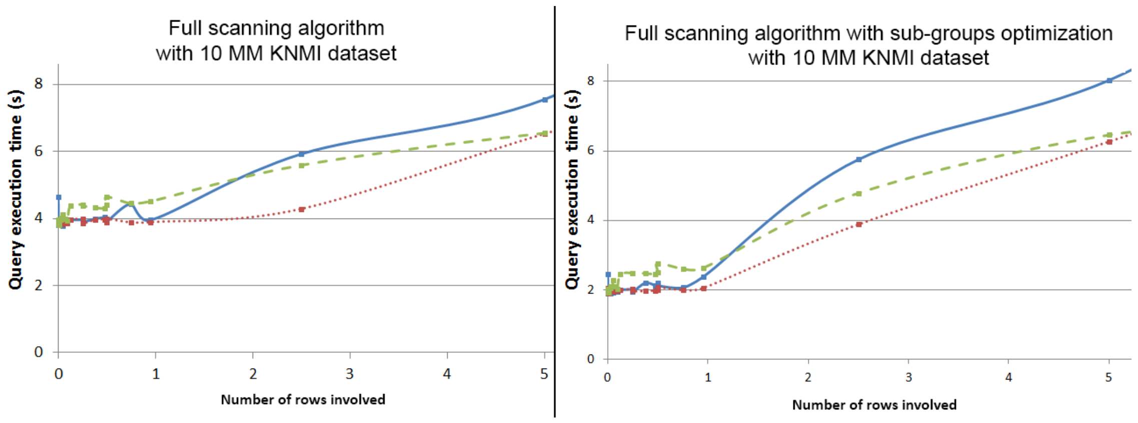


Figure 6.11: Test results of the full scanning algorithm and the sub-groups optimization

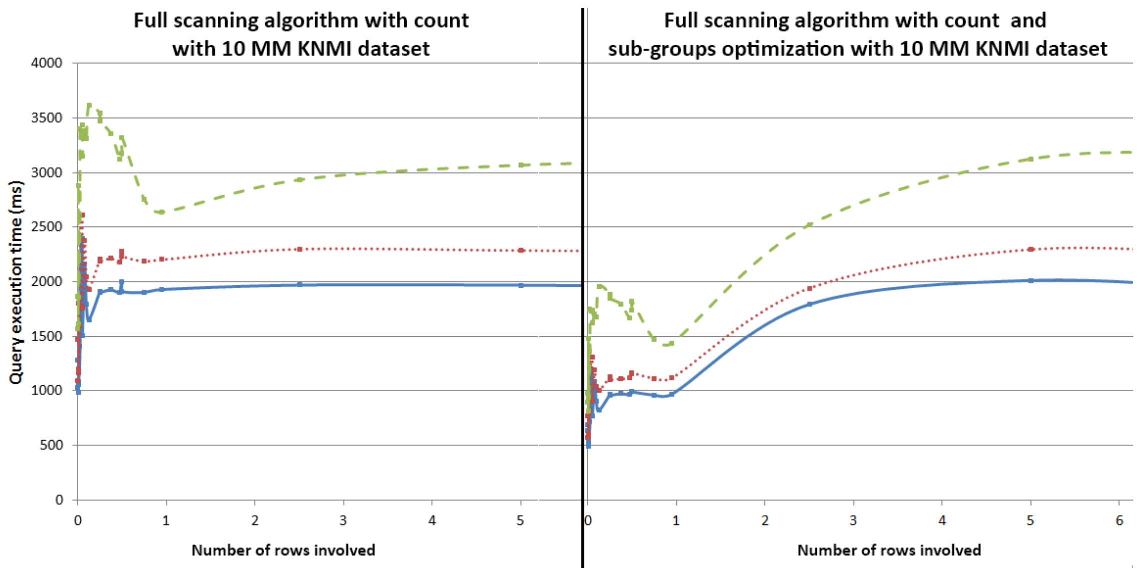


Figure 6.12: Test results of the full scanning with count algorithm and the sub-groups optimization

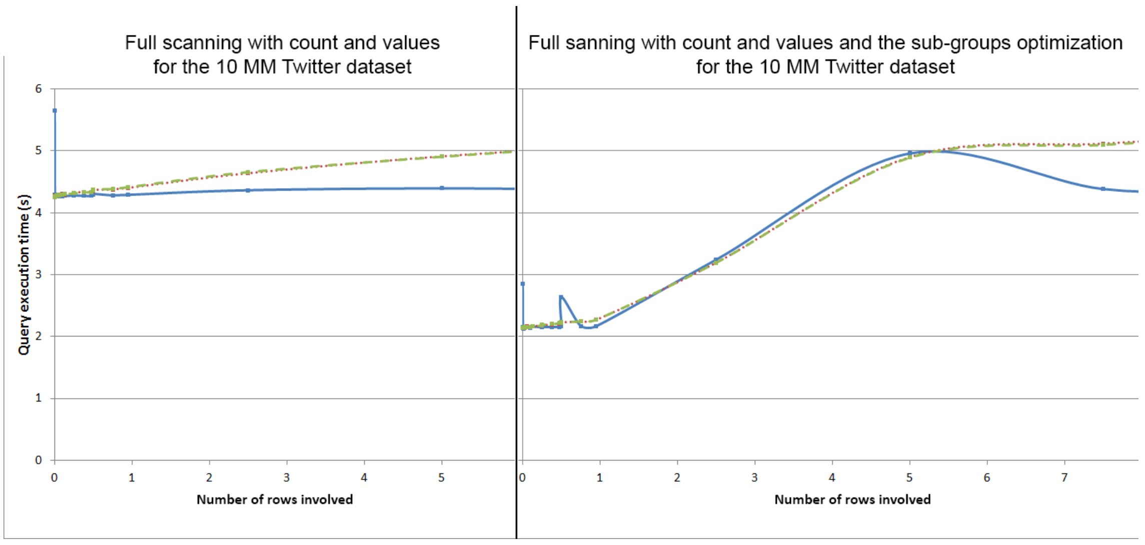


Figure 6.13: Test results of the full scanning with count, values and sub-groups optimization

This section has discussed, analyzed and evaluated the effects of each algorithm and optimization on the query performance, the relationship between the query performance and the number of tuples involved in a query, the time complexity of this relationship and the relationship between the total size of the dataset and the query performance. Test results have clearly shown that the algorithms, variants and optimizations that have been developed have a very mixed effect on the query performance.

The next section will analyze the storage requirements for each algorithm and optimization, which is an important part of the evaluation of this research because there is a very high dependence on the use of pre-computed data structures and streams to improve the query performance.

6.2 Storage requirements

This section will evaluate the storage required by each alternative algorithm and optimization necessary to improve query performance of median (aggregation) queries. The storage requirements of each algorithm and optimization will be discussed separately in the following subsections, starting with the basic algorithm.

6.2.1 Basic algorithm

The basic algorithm has no (additional) storage requirements besides the original dataset and its related indexes (i.e. a B-tree index on the dimensions of the dataset) because it does not use any pre-computed data to improve query performance; only the original data is used to calculate the median.

6.2.2 Shifting algorithm

The shifting algorithm depends on the use of two types of pre-computed data structures: (i) left- and right shift factors for each tuple per dimension and (ii) left- and right binary data streams. Table 6.1 lists the storage requirements for both types of structures in a percentage of the total size of the original data set for both the PEGEL and the KNMI datasets.

Dataset	Size of data streams	Size of shift factors	Total size
PEGEL	6.5%	9.5%	16.0%
KNMI	15.0%	9.5%	24.5%

Table 6.1: Storage requirements of the shifting algorithm

The figures in table 6.1 show that the additional storage needed by the shifting algorithm lies between approximately 15% and 25% of the original dataset. This algorithm needs less storage for the PEGEL dataset than the KNMI dataset because the dimension values, which must be encoded in the data streams, are also used as the unique key of each tuple due to their uniqueness property (i.e. only 1 tuple at each specific time). This is not possible for the KNMI dataset since multiple tuples have the same time value thereby requiring the inclusion of an additional (artificial) tuple key into the data streams.

The figures for the size of the shift factors are somewhat misleading, because it seems like the size of the shift factors are the same irrespective of the number of dimensions. This is only the case for the KNMI dataset because, although is has twice the number of dimensions as the PEGEL dataset, the second dimension of the KNMI dataset (the station dimension) has a very limited number of distinct values, i.e. only 10 unique values, and therefore only a very limited number of shift factors for this dimension. This is not the case for datasets with multiple dimensions, each with a large number of distinct values, whereby the storage size of the shift factors could increase greatly for each dimension. This is another reason why the shifting algorithm is less suitable for datasets with multiple dimensions.

The next subsection will evaluate the storage requirements of the full scanning algorithm.

6.2.3 Full scanning algorithm & with count variant

The full scanning algorithm only depends on a single pre-computed data structure: the pre-sorted binary data stream. Table 6.2 lists the storage figures for the binary data streams for the test data sets as a percentage of the original data set. These figures show that the storage required by this algorithm is quite substantial although it is still surprisingly small, since the binary data streams are almost an exact copy of the original dataset except for the actual measure value. Clearly by storing the data as a binary stream the overhead of the database engine is saved in storage space.

Dataset	Size of data streams
PEGEL	6.5%
KNMI	15.0%
Twitter	30.0%

Table 6.2: Storage requirements of the full scanning algorithm

The variant of the full scanning algorithm whereby the count is available before query time does not require additional storage space and uses the same binary data streams as the basic full scanning algorithm. However this variant depends on the availability of an efficient count operation, such as the Stairwalker algorithm, which has its own storage requirements, most likely in the form of additional pre-computed data structures (i.e. pre-aggregates). Concrete figures for this are not available.

The next subsection will evaluate the storage requirements of the optimization technique whereby the values are included in the binary data streams.

6.2.4 Including values optimization

This optimization is based on including the actual measure values into the binary data streams for both the shifting and full scanning algorithms. The previous two subsections have shown that the binary data streams require the same size for both algorithms therefore they can considered to be the same for the analysis into the storage requirements of the values optimization.

Table 6.3 shows the storage sizes of the binary data streams with the values included for each data set. The most interesting about these figures is the fact that the size of the streams for the KNMI and Twitter data sets has actually decreased, despite the fact that they now include the actual measure values. This seems contradictory but has a very logical explanation: the actual value has replaced the tuple keys in the data streams. Clearly, these tuple keys need more space than the actual values thus resulting in a decrease in storage space if these keys are replaced with the actual values. For the PEGEL dataset the size of the data streams slightly increase because for this data set the values do not replace the tuple keys, since these were never part of the data stream.

Dataset	Size of data streams
PEGEL	8.0%
KNMI	8.0%
Twitter	24.5%

Table 6.3: Storage requirements of the including values optimization

The binary data streams whereby the value is included allow the median to be calculated without the need for the original dataset, thereby the original dataset does not need to be available. This means that the original dataset can remain somewhere external, e.g. in the data warehouse, whilst the binary data streams are updated nightly and used by the end-user to explore the data set. This means that this technique can be used as an compression technique, since the binary streams are significantly smaller than the original data sets.

The next subsection will analyze the storage requirements of the sub-groups optimization.

6.2.5 Sub-groups optimization

The sub-groups optimization is based on generating additional data structures of smaller parts of the full dataset which is then used to calculate the median for smaller queries. Concretely this means for the shifting algorithm that for each additional level of sub-groups new data streams and shifting factors must be calculated and stored. For the full scan algorithm only the additional data streams must be stored.

Hence for each additional level of sub-groups the storage requirements as discussed in the previous subsections increase linearly. For example, if an additional level of sub-parts is generated for the PEGEL data set for the shifting algorithm the storage requirements increase from 16% of the full dataset to 32% of the full dataset. This also implies that the best algorithm to use the sub-groups optimization with is the algorithm that uses the least amount of storage space which is the full scanning algorithm with the values optimization.

Since this optimization is only able to improve the performance for smaller queries and the storage requirements are quite steep, especially compared to the benefits it provides, a very careful consideration must be done before employing this optimization to determine if the advantages outweigh the disadvantages. Analysis of the frequency of smaller queries and the performance of the basic algorithm with regards to smaller queries is necessary before this optimization should be used.

This subsection finishes the evaluation into the storage requirements of the algorithms and optimizations that have been developed. The next section will provide a summary of the evaluation.

6.3 Summary

This section summarizes the outcome of the evaluation of the developed algorithms and optimizations. Evaluation has been done on two criteria: (i) query performance, with a focus on the time complexity of the performance and (ii) storage requirements. The evaluation has shown that:

- The shifting algorithm is the best algorithm to improve the query performance and provides the best scalability with regards to the query performance
- For highly dimensional data sets the best alternative is the count variant of the full scanning algorithm although this algorithm has less effect than the shifting algorithm
- The optimization whereby the values are included into the data streams has a mixed effect on the query performance
- The sub-groups optimization also has little effect on the query performance or its scalability but is able to improve the performance of a (subset) of smaller queries
- The shifting algorithm needs more storage than the full scanning algorithm because the shift factors must also be stored
- The optimization to include the values is able to reduce the storage space needed by the algorithms and makes it possible to process median queries without the original dataset
- The sub-groups optimization has a high storage cost when compared to the limited improvement in query performance and therefore should be very carefully considered before use.

Figures 6.14-6.16 show a comparison of the query performance for each algorithm/optimization (except the sub-groups optimization) for a median query of 9.5 million rows for each of the three data sets. These charts show the difference in query performance improvement of each algorithm for each data set. Clearly the shifting algorithm is able to provide the best improvement in query performance. But these charts also confirm that the best alternative is the count variant of the full scanning algorithm, since this algorithm is still able to significantly improve the query performance, as clearly visible in the figures. The charts also corroborate the fact that the optimization whereby the values are included has a mixed effect on the query performance; in some cases it is able to improve the query performance and in other cases it actually degrades the performance.

The next section will provide a conclusion of the evaluation of the algorithms and optimizations.

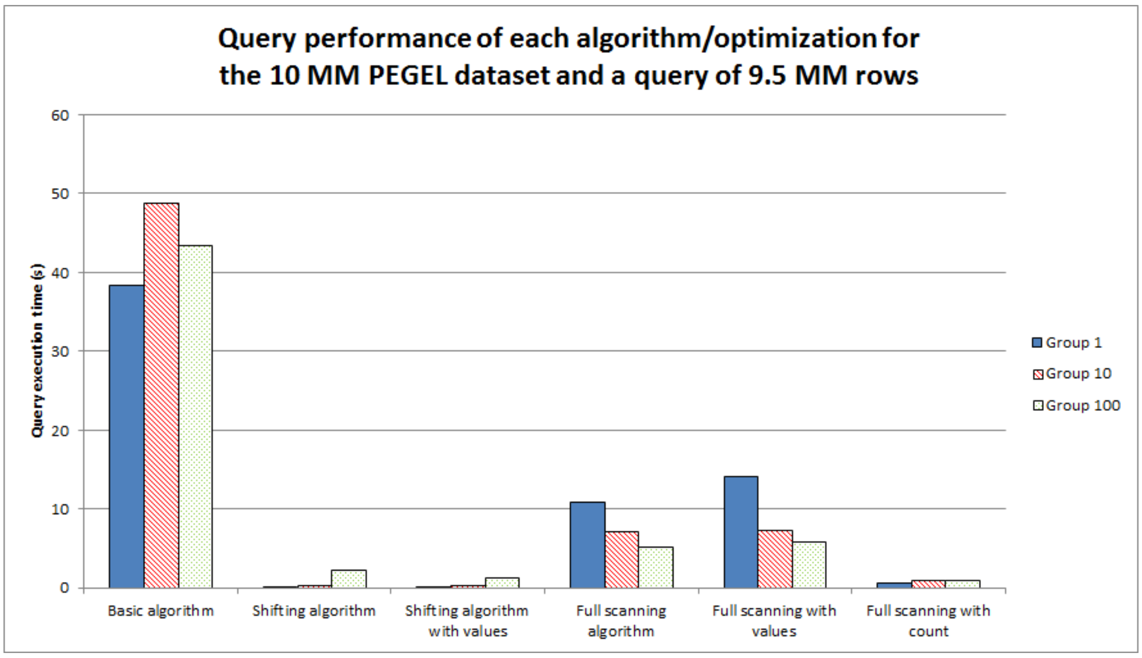


Figure 6.14: Query performance comparison for the PEGEL data set

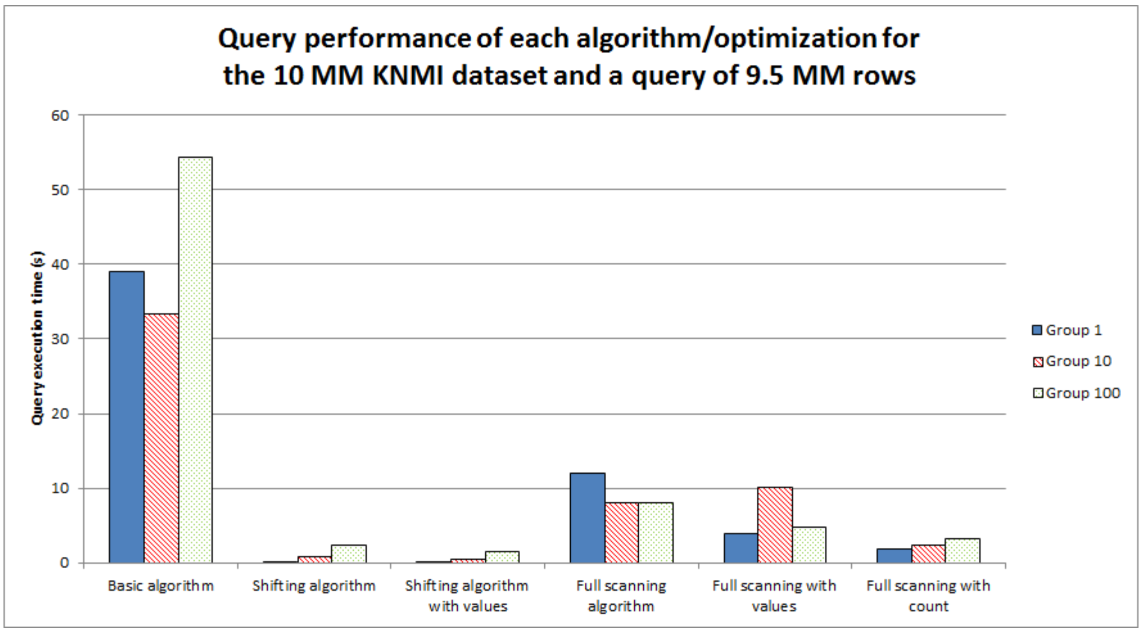


Figure 6.15: Query performance comparison for the KNMI data set

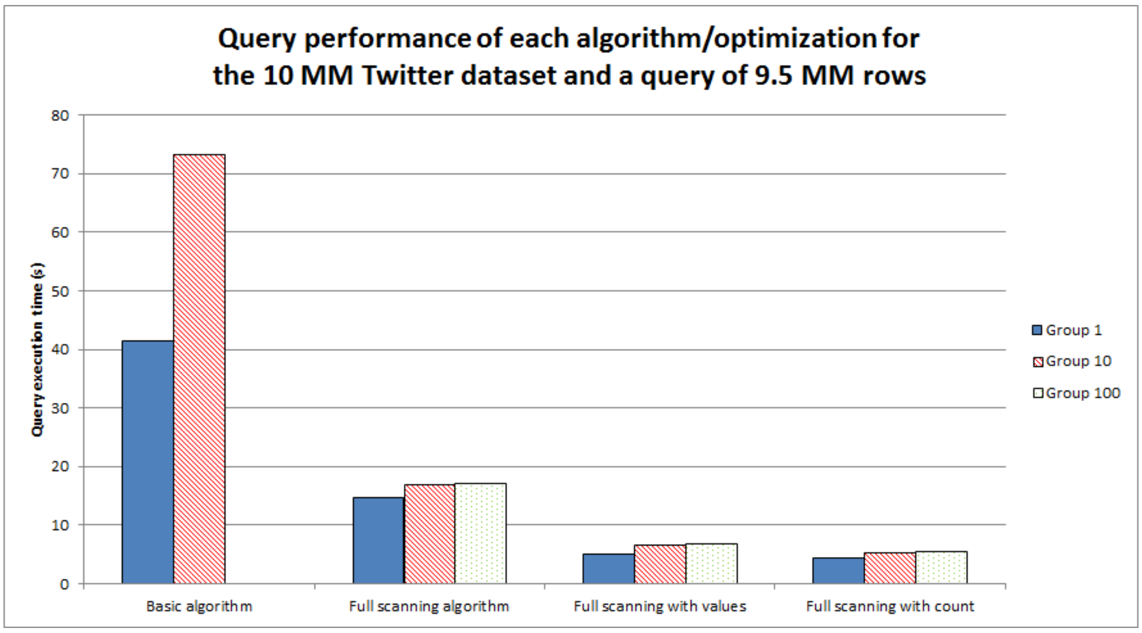


Figure 6.16: Query performance comparison for the Twitter data set

6.4 Conclusion

In this research several alternative algorithms and optimizations have been developed that are designed to improve the query performance of median (aggregation) queries. This chapter these algorithms and optimizations have been evaluated on two points: actual improvement to the query performance with a focus on the improvement to the time complexity of the query performance instead of actual absolute query performance and the storage requirements. All evaluation has been based on actual test results, done with the experimental setup whereby all external variables have been kept constant.

First, the query performance of each algorithm has been evaluated. Careful analysis of the basic algorithm has shown that there is a clear linear relationship between the number of tuples involved in a median (aggregation) query and the query performance. This relationship is distinctly visible in the graphs of the test measurements.

Next, test results of the shifting algorithm were analyzed. Since this algorithm is not very suitable for multi-dimensional data sets evaluation of this algorithm could only be done for the PEGEL and KNMI data sets. Analysis has shown that this algorithm is able to greatly improve the query performance and completely remove the relationship between the query performance and the number of tuples affected by the query. Instead there is now a linear relationship between the total size of the full dataset and the query performance. Since this linear relationship has a more gradual growth-rate it is able to scale better than the original linear relationship of the basic algorithm.

Test results for the shifting algorithm also showed that the query performance noticeably decreases as the number of groups of a query increases at a linear pace. For a query with many groups (i.e. nr. of groups $>>$100) this could have a large negative effect on the query performance.

Then, the test results of the basic variant of the full scanning algorithm were evaluated, which showed that, although the absolute query performance was almost 10x better than the basic algorithm, there is still a similar linear relationship between the number of affected tuples and the performance of a query. Hence, this algorithm does not improve the scalability of processing median queries.

After that the count variation of the full scanning algorithm was evaluated. The rest results of this algorithm show similar effects as with the shifting algorithm, i.e. the relationship between the number of affected tuples and the performance of a query no longer exists and a new relationship has arisen between the size of the total dataset and the query performance. For this algorithm however the linear relationship between the size of the total dataset and the query performance is steeper than the shifting algorithm. This means that the scalability of this algorithm is less than the shifting algorithm. It is very clear however that this variant of the full scanning algorithm is able to deliver much better query performance than the basic variant. It must not be forgotten however that this variant depends on the availability of a very efficient count operation and that this also has a performance penalty, unlike the shifting algorithm.

Then, the two optimizations were evaluated to determine their effect on further optimizing the query performance. Test results have shown that the with-values optimization has a very mixed effect on the query performance and the time complexity. In some cases the time complexity and/or query performance is significantly improved and in other cases it is actually worsened. The sub-groups optimization is able to improve the performance of smaller queries however its effect is limited to very small queries, as demonstrated by the test results.

The storage requirements of each algorithm and optimization have also been analyzed.This has shown that the shifting algorithm needs at least 15% additional storage space to store its pre-computed data structures and that this figure increases as the number of dimensions increases, which is another indication that the shifting algorithm is less suitable for highly dimensional data sets.

Both variants of the full scanning algorithm require less storage space than the shifting algorithm because it does not require storing shifting factors in addition to data streams. The results have shown that this algorithm needs up to 30% additional storage space although this figure can be less (e.g. for the PEGEL dataset it is only 6.5%).

Most surprisingly were the results of the analysis into the store requirements when applying the optimization whereby the values are included into the data streams. This optimization actually reduces the storage space that is needed in most cases, depending on the size of the actual value and the tuple keys. In addition, this optimization makes it possible to process median (aggregation) queries using only the data streams thereby avoiding the need to actually have the full dataset available online. This could be especially useful in OLAP contexts whereby original data sets are commonly maintained separately in the data warehouse and the front-end application is served by subsets of data (cubes).

This chapter also looked at the storage requirements of the sub-groups optimization. Investigation has showed that this optimization has a linear relationship between the amount of storage it requires and the number of additional levels that are generated for the sub-groups of data, since for each additional level the full data set is again pre-processed into data streams and shift factors (for the shifting algorithm) albeit into *n* sub-parts but this has no direct effect on the storage required. Therefore this optimization has a high cost for potentially little effect, since it is only able to improve the performance of smaller queries and each additional level has a decreasing return on (storage) investment because as the sub-parts get smaller the amount of queries matching a sub-part also gets smaller.

Finally this chapter with a summary of the most important insights that the evaluation of the algorithms and optimization has provided. Based on these insights a framework has been developed for selecting the best algorithm and/or optimization in various circumstances. This is discussed in further detail in the next chapter.

Chapter 7

Algorithm selection framework

This chapter will outline the work that has been done into developing a framework that can be used to determine which algorithm and optimization will provide the best query performance for different circumstances. This framework has been developed as part of the research into the alternative median algorithms and the evaluation of these algorithms and related optimizations.

The next section will discuss how the framework has been developed and provide the actual framework. Then, the framework will be evaluated and its strengths and weaknesses discussed. Finally this chapter will end with a conclusion.

7.1 Development of framework

This section will first outline the most important factors that have driven the development of the algorithm selection framework. Then the framework will be provided and with an explanation on its usage.

Based on the insights of the evaluation of the alternative algorithms and optimization in chapter 6 and summarized in section 6.3, the following factors have to been determined to play a role when choosing which algorithm or optimization should be used:

- How many dimensions does the data set have?
- Is there an efficient count operation available?
- Is the storage size of the actual measure value smaller than the storage size of the artificial tuple key?
- Does the expected query work load contain many queries that are grouped into significantly more than 100 groups?
- Does the expected query work load contain many smaller queries?

Based on these factors a decision tree has been developed which encapsulates the different possibilities into a single framework. This can be seen in figure 7.1.

The selection tree consists of three distinct selection phases. The initial phase, *start situation*, consists of the basic algorithm, which works in every circumstance but does not offer any improved query performance. The next phase, *algorithm selection*, involves the selection of the correct algorithm for improved query performance. The first decision in this phase is determining the number of dimensions of the data set. If the number of dimensions is two or less test results have shown that the shifting algorithm should be used for the best query performance. If the number of dimensions is more than two however then the decision tree points to either the full scanning algorithm or the full scanning with count algorithm, depending on the availability of an efficient count operation. Test results have shown that the full scanning with count algorithm has the best performance of the two.

The next phase, *optimization selection*, determines which additional optimization should be selected for further improving query performance and/or reducing storage requirements. The first decision in this phase is to determine if the storage size of the actual measure values is smaller than the storage size of the (artificial) tuple keys or if the expected work load is expected to have many queries whereby the number of groups is significantly bigger than 100. In the first case the required storage space can

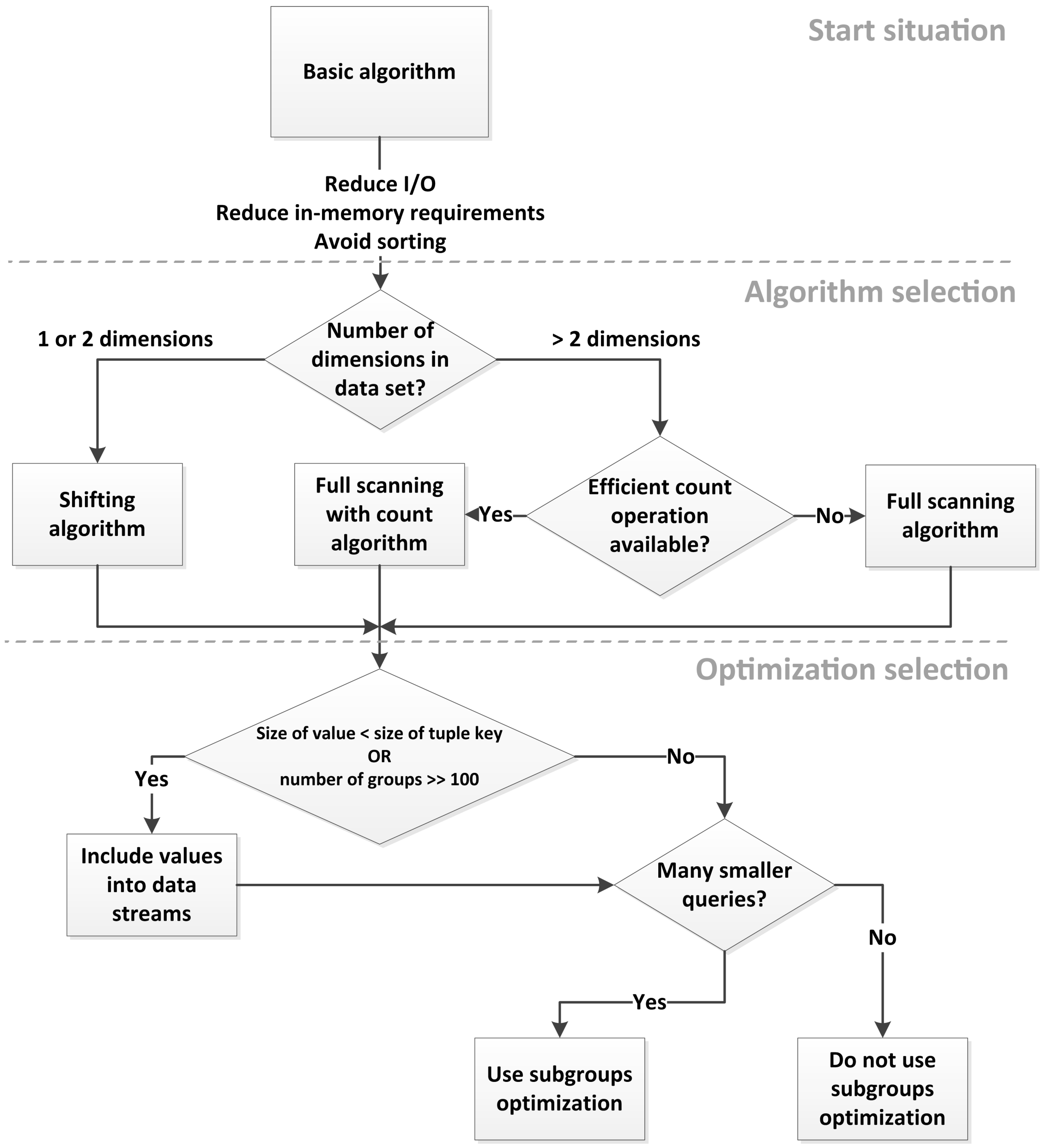

Start situation
Basic algorithm
Reduce I/O
Reduce in-memory requirements
Avoid sorting
Algorithm selection
Number of dimensions in data set?
1 or 2 dimensions
> 2 dimensions
Shifting algorithm
Full scanning with count algorithm
Yes
Efficient count operation available?
No
Full scanning algorithm
Optimization selection
Size of value < size of tuple key
OR
number of groups >> 100
Yes
No
Include values into data streams
Many smaller queries?
No
Yes
Use subgroups optimization
Do not use subgroups optimization

Figure 7.1: Algorithm selection decision tree

be reduced by including the values and in the second case test results have shown that by including the values the query performance is improved when dealing with queries with many groups. If neither case is true then there is no real benefit in using the values optimization and therefore should not be used.

The second decision in this phase is to forecast the number of expected small queries that cover only very small parts of the complete data set. Since the test results have shown that the developed algorithms tend to have worse query performance for very small queries the sub-groups optimization is used to overcome this drawback. However the test results have also shown that this optimization is only suitable for very small queries and has a high cost. Therefore only a very large number of small queries make this optimization feasible. If the expected workload does not have this property then this optimization should not be selected.

With this selection tree the correct algorithm and optimizations are easily selected for every circumstance. The next section will evaluate the strengths and weaknesses of this selection tree.

7.2 Evaluation

This section will briefly discuss the strengths and weaknesses of the algorithm selection framework, as outlined in the previous section. First the benefits of the selection tree will be covered and then the weaknesses and outstanding issues.

Based on the research into the development and usage of the selection tree it has been established that it offers the following benefits:

Usable in every circumstance / data set
The decision tree is suitable for *every* circumstance irrespective of the specific data set and/or expected work load. By starting with the basic algorithm, which already offers a solution for every circumstance, and systematically moving onto several alternative algorithms/optimizations a suitable combination of algorithm and optimization is always available. This means that query performance can always be improved for every circumstance.

Easy to select correct algorithm/optimization
The selection tree is very concise and small, making it easy to follow and select the correct algorithm/optimization. By keeping it simple the likelihood of actually using the decision tree is much higher as opposed to a very complex framework, which is likely to remain unused in many circumstances.

Based on actual test results
The decision tree is based on the experimental tests results of this research. Therefore the decision tree has a strong empirical backing and guarantees similar performance improvements if correctly used.

In addition to the above benefits the current version of the selection tree still suffers from a few potential issues and drawbacks, primarily due to the lack of research into specific problems. These are:

Depends on a priori knowledge of the expected work load
Several decision splits in the selection tree depend on a priori knowledge of the expected query work load, e.g. does it have many smaller queries or many queries with a high number groups. In some cases this a priori knowledge may not be available or may not be an accurate prediction of the future work load therefore this is a weakness in the decision tree and may lead to the selection of an optimization which has a negative effect on the query performance or storage requirements. Fortunately this dependence only occurs in the optimization selection phase of the tree and has no influence on the selection of the actual algorithm.

Unclear on number of groups for with-values optimization
At present the decision tree has a decision split whereby the split (partly) depends on whether the number of groups (that queries are grouped into) is significantly bigger than 100. However there is no clear definition of significantly bigger and this is left up to the readers interpretation. Clearly this is a weakness in the decision tree but unfortunately a thorough investigation into this area has not been part of this research thus a more accurate number cannot be provided.

Unclear how many smaller queries are needed for sub-groups optimization
Similar to the previous drawback, the decision tree has a decision split which depends on whether there are many smaller queries in the expected query work load. However a clear definition of ”many smaller queries” cannot be given because the test results do not support a more accurate definition. An extensive investigation into this specific phenomena is likely to lead to a more accurate definition. This weakness may lead to significantly increased storage requirements or decreased query performance if the incorrect decision is made due to uncertainty.

Although the decision tree clearly has some unresolved drawbacks these are concentrated in the optimization selection phase of the selection tree. Given this, its strengths outweigh its problems and offers a good framework for selecting the most optimal algorithm and optimization in every circumstance.

The next section will provide a conclusion to this chapter.

7.3 Conclusion

This chapter introduced a framework for selecting the right algorithm and optimization for improving query performance for different circumstances. This framework depends on several factors that are based on the research into and evaluation of alternative median algorithms and optimizations.

These factors have lead to the development of a decision tree, which is separated into three distinct phases: the start situation, the algorithm selection and lastly the optimization selection. The start situation consists of the basic algorithm, which is suitable for every circumstance and can always be used but offers no improvement in query performance. The algorithm selection phase leads to the correct selection of an algorithm and improved query performance. Finally the optimization selection phase is used to select additional optimizations which can further improve query performance or decrease storage requirements.

The decision tree offers several benefits: ease of use in selecting the right algorithm and optimization(s) for every circumstance and a solid empirical backing. However the current version has some drawbacks in the optimization selection phase whereby there is a dependence on a priori knowledge of the expected query work load and vague definitions of certain crucial factors influencing the selection of optimizations. Further research is likely to reduce or eliminate these drawbacks.

Given this information, the current algorithm selection framework can reliably be used to determine the correct algorithm and optimizations for different circumstances.

The next chapter will conclude this thesis by answering the research question and subquestions, discuss potential issues that may exist with this research and outline possible future research directions.

Chapter 8

Conclusions

This chapter provides a conclusion to this thesis into improving query performance for holistic aggregate queries for real-time data exploration, whereby there has been a focus on the median aggregation function. The first section will provide a conclusive answer to the research question and related subquestions. The second section will analyze the work that has been done and outline some potential threats to the validity of the research. Finally the last section will discuss possible future directions of research.

8.1 Research Questions

This section will provide an answer to the main research question and subquestions. First the subquestions of the research will be answered in the following subsections. Then, this section will finish by answering the main research question.

8.1.1 What factors influence the query performance?

The theoretical investigation into the factors that have an influence on the query performance has shown that there are two factors that have the biggest influence on the performance: (i) disk seek time and (ii) disk bandwidth. Disk seek time is the time that it takes for a hard drive to physically move the disk head to the correct location of the desired data. Disk bandwidth is the amount of data that can be transferred per time unit, i.e. megabytes per second. Other factors, such as CPU clock speed, (amount of) memory have a much smaller influence on the performance of a query than disk operations [1]. A complete discussion of the factors related to query performance can be found in chapter 3.

When considering a median query that covers a large range it is likely to involve many individual tuples. During this research it has been shown (in section 5.1) that actual seek time is not a big influence for types of queries because, with the correct clustering of the dataset, the tuples can be fetched from disk in one continuous scan. Therefore the query performance is partly determined by the time it takes to fetch all the tuples from disk, i.e. the amount of disk bandwidth available.

Tests results have shown that in-memory operations also have an influence on the performance of the query. Generally in-memory operations are considered to be several orders faster than disk operations however the performance cost of in-memory operations cannot be completely disregarded, especially when dealing with large datasets

Therefore, in the scope of this research, it has been determined that the following factors have the biggest influence on the performance of median (aggregation) queries:

- Disk bandwidth, i.e. the time it takes to fetch all tuples from disk
- Speed of in-memory operations

Section 3.3 outlines the empirical investigation into these factors whereby it has been shown that they indeed have the biggest influence on the performance of database queries for the database engines that have been tested.

The next subsection will provide a conclusion to the second subquestion.

8.1.2 Which algorithms improve query performance?

Research into existing algorithms used by database engines, as further outlined in section 2.3, has shown that most database engines use a **basic algorithm** to process median queries. This algorithm has been analyzed and its query performance and scalability tested, which has lead to the conclusion that this basic algorithm offers poor performance and does not scale well primarily because of two reasons: (i) *all* tuples must be fetched from disk to be able to calculate the median and (ii) *all* tuples must be stored and sorted in-memory whereby especially the sorting has a high performance cost. More discussion about this basic algorithm can be found in section 5.1 and test results of this algorithm are found in section 6.1.1.

To solve these two problems an alternative algorithm called the **shifting algorithm** has been developed. This algorithm is able to process median queries without needing to fetch all tuples from disk or use in-memory operations. This algorithm depends on the use of pre-computed, i.e. before query-time, binary data streams which contain all the data in a pre-sorted state and pre-calculated shift factors for each tuple in the dataset. With this extra data the shifting algorithm is able to determine the location of the median of a query and quickly scan towards the correct tuple, thereby returning an exact result. More details about the shifting algorithm can be found in section 5.2.

Test results of this algorithm, outlined in more detail in section 6.1.2, have shown that it is indeed able to greatly improve the performance of median (aggregation) queries and improve the scalability of such queries. Due to its method of operation this algorithm is less suitable for datasets with multiple dimensions. To solve this problem another alternative algorithm has been developed, namely the full scanning algorithm.

The **full scanning algorithm** is a simpler algorithm that is also suitable for higher dimensional data sets. This algorithm relies on the use of a single pre-computed binary data stream that contains the full data set in a compact pre-sorted state. This offers two advantages in processing a median aggregation query: (i) the data no longer needs to be sorted thereby reducing in-memory operations and (ii) the binary data streams are much compacter than the original dataset and is therefore much faster to read from disk. More details of this algorithm can be found in section 5.3.

Two variants of this algorithm has been developed: a basic variant whereby the full data stream must always be scanned and all the tuples matching the query or current group (when processing a group-by/aggregation query) must be stored in memory to find the median and an improved variant whereby the total number of tuples affected by the query is available before processing avoiding the need to store any tuples in memory and drastically reducing the full scanning time. This second variant, called the **full scanning with count algorithm**, relies on the availability of a very efficient count operation, such as the count operation made available by the Stairwalker algorithm developed at the University of Twente (as discussed in section 2.4.1). Section 5.3.1 covers this second variant in more detail.

Test results with the two variants of the full scanning algorithm, as discussed in sections 6.1.3 and 6.1.4, have shown that both variants greatly improve the query performance however the basic variant still demonstrates a linear relationship between the query performance and the number of tuples affected by a query which means that the scalability of the algorithm is the similar as the basic algorithm. The second variant of the full scanning algorithm however shows much more promise with regards to scalability and is able to offer a fixed query performance irrespective of the number of tuples affected by a query although query performance is now linearly related to the full size of the dataset.

Two additional optimizations have been developed that can be used in conjunction with the shifting algorithm and both full scanning algorithm variants. One optimization, called the **with values optimization**, relies on including the actual measure values into the binary data streams instead of only including only the dimensions and a tuple key. This has the potential to reduce the amount of additional lookups when actually calculating the median after the correct tuple key(s) for the median have been determined by the algorithms. Test results have shown that this optimization has a mixed effect on the performance of aggregation queries; in some cases it significantly improves the performance and in other cases in may actually deteriorate the performance. See section 6.1.5 for more details.

The second optimization, called the **sub-groups optimization**, is used to improve the performance of queries with a small range that only cover a small part of the total dataset. Test results have shown that these types of queries potentially have worse performance with both the shifting and full scanning algorithm when compared with the basic algorithm. By employing the sub-groups optimization this effect is reduced. It relies on generating additional pre-computed binary data streams and shift factors (for the shifting algorithm) for smaller parts of the full data set, for example an additional set of streams could be generated for the first and second half of the full dataset separately. These would then be used to process smaller median queries and provide improved query performance due to the decreased scanning and shifting times. This optimization is discussed in more detail in section 5.4.

Test results have shown that the effect of this optimization is very strictly limited to smaller queries. Although the effect is very clearly visible, it has a high storage cost (i.e. each additional level linearly increases the amount of storage necessary) and aggregation queries are quickly too big or do not fit exactly into a sub-group. Therefore very careful analysis of these types of queries is necessary before being able to successfully employ this optimization. Section 6.1.6 discusses the effect of this optimization in greater detail.

This subsection has provided an answer to the second subquestion whereby several alternative algorithms for processing median queries and two possibly further optimization techniques have been outlined. Each algorithm and optimization has its own pros and cons, hence it is important to have a good understanding when each algorithm and optimization is able to provide the best performance and scalability improvement. This will be covered in the next subsection.

8.1.3 Which algorithms are best suitable for different circumstances?

The answer to this subquestion lies in the algorithm selection framework which has been developed as part of this research. This framework, covered in more detail in chapter 7, is based on the test results of the algorithms and optimizations that have been developed and evaluated. This has resulted in a list of several key factors which should be considered when selecting the correct algorithm and optimizations for a specific circumstance. The specific factors can be found in section 7.1.

Based on these key factors a decision tree has been developed which is encapsulates these factors and can be used as a framework for selecting the correct algorithm and optimizations for a specific circumstance. This decision tree can also be found in section 7.1, together with an explanation on how to use this tree.

Evaluation of the decision tree has shown that it is usable for every type of circumstance and/or data set and provides an easy-to-use framework for selecting the correct algorithm and optimizations. At present however it does have some unclear definitions of certain factors which may lead to an incorrect algorithm selection. Further details of the strengths and weaknesses of this tree can be found in section 7.2.

The next subsection will present the conclusion to the main research question.

8.1.4 Main research question

This subsection will provide a clear and definitive conclusion to the main question of this research. The main research question is as follows:

> *How can the performance of queries with the median aggregate function be improved whilst still returning accurate results?*

Research has shown that the query performance of queries with the median aggregate function is largely dependent on two main factors: disk I/O operations and in-memory operations, and therefore to improve the performance of such queries these factors must be minimized. The basic algorithm for processing median queries, as used by most current database engines, is largely dependent on these performance factors and thus has very poor query performance and does not scale well.

To reduce the impact of these performance factors several new algorithms and optimizations have been developed that are still able to compute the median and return an exact result. These algorithms are primarily able to do so by making extensive use of pre-computed additional data structures, such as extra database columns with information and supplementary binary data streams. Three alternative algorithms, each with their own pros and cons have been developed. These are **the shifting algorithm**, **the full scanning algorithm** and **the full scanning with count algorithm**. In addition, two optimizations for further performance improvement have been developed. These are **the with values optimization** and **the sub-groups optimization**. More comprehensive discussion about the development of these algorithms and optimizations can be found in chapter 5.

Experimental tests, which are fully discussed in chapter 6, have shown that the shifting algorithm is the best algorithm for two reasons: (i) it is able improve the query performance the most and, even more important (ii) it is able to increase the scalability of the query performance the best. This algorithm removes the linear relationship between the query performance and the number of tuples involved in a query, as is the case for the basic algorithm, and changes this into a linear relationship between the query performance and the size of the full dataset. This does have a possible negative effect on very small queries (i.e. covering a small number of tuples) but a positive effect on larger queries.

Unfortunately the shifting algorithm is less suitable for data sets with multiple dimensions due to its complexity. For multi-dimensional data sets the full scanning algorithm is better suited and able to improve query performance. The basic variant of this algorithm has a similar relationship between number of affected tuples and query performance as the basic algorithm and is therefore able to improve the scalability less than the shifting algorithm. However the count variant of this algorithm has a similar time complexity as the shifting algorithm.

The two optimizations can be used in conjunction with all the algorithms and offer an additional query performance improvement but do very little for the scalability of such queries. However the *with values optimization* is very suitable to be used in OLAP contexts, because this optimization allows the algorithms to process median queries without the use of the full dataset, only requiring the pre-computed data structures. As such, these pre-computed data structures can act as disposable data marts for the end-user to explore the data.

Finally an algorithm selection framework has been developed to determine which algorithm and optimization is best suitable for improving query performance and scalability for different circumstances and data sets. This framework provides a mechanism, in the form of an easy-to-use decision tree, for selecting the right algorithm and optimizations for improving the performance of queries with the median aggregate function.

The next section will discuss potential threats to the validity of the results of this research.

8.2 Threats to Validity

This section will discuss some potential issues that may be potential threats for the validity of this research and the conclusions. There are several issues that could possibly have such an influence on this research that the conclusions are not longer trustworthy. These are the following issues:

Unexpected caching
In this research, issues related to caching and buffering by the database engine used in the experimental tests has been taken into account, however it is still conceivable that some caching has managed to occur thereby influencing the results of the tests. Considering the results however there is no indication that caching has had a big influence.

External load on the primary test platform
The primary test platform for the experimental tests is a computer maintained by the University of Twente and used by several members of the Database Group. Hence, there was no exclusive access to this platform, making it possible that additional load was generated by another person during one or multiple of the experiments. This could have had an influence on the tests results of these experiments however there is no way to determine if this has happened. Since the experiments have been ran multiple times over a longer period any erroneous results due to external load have likely been discarded.

Optimized for specific data sets
It is possible that query performance has been improved specifically for a certain data set or type of data sets. The likelihood of this happening has been minimized by using 3 distinctly different data sets in the experimental tests, each with their own characteristics. However it is possible that there may have been an optimization towards a specific situation and this may surface at a later stage when the results of this research are used more widely in a production environment.

Optimized for a specific database engine
Similar to the previous problem, it is very possible that the alternative algorithms are very specifically tailored towards the database engine used in the experiments (i.e. PostgreSQL). From a theoretical standpoint for example a (partially) in-memory database such as MonetDB is unlikely to benefit as much as PostgreSQL from the binary data streams technique, which is heavily used by the algorithms, since all the data is generally kept in memory already anyway by those types of databases. However considering the test results it is very likely that this result will still have a positive impact on the query performance but on a smaller scale.

The above issues have been identified as the main problems that could weaken or undermine the validity of this research. However, considering the additional precautions undertaken during this research it is likely that these potential risks have been minimized or eliminated.

The next section will discuss possible future research directions.

8.3 Future work

This section discusses some possible directions for future researched based on the work that has been done in this research, which has laid the groundwork for further investigation into optimization median aggregation queries but also other types of holistic aggregation queries. The following research directions are most worth exploring further:

Implementation/testing with other database engines
It would be very interesting to spend additional effort into implementation prototype versions of the algorithms and optimizations for other database engines (e.g. MySQL, SQL Server, Oracle, MonetDB, etc) and re-running the experimental tests to determine the influence with other database engines. This would remove one potential threat to the validity of this research, as discussed in the previous section, and provide a greater understanding of the potential for improving query performance of each algorithm and optimization.

Commercial/production implementation
An actual commercial/production-worthy implementation of one or several of the algorithms would allow for further research with more data sets and additional use-cases. It is likely that this would lead to new insights into better ways of improving query performance for median (aggregation) queries.

More investigation into the shifting algorithm with regards to multiple dimensions
This research has shown that the shifting algorithm is the best algorithm for improving query performance and scalability however it was not possible to fully apply this algorithm for data sets with multiple dimensions. Hence, more investigation into this specific algorithm and its applicability towards higher-dimensional data sets shows much promise.

Applying the results towards other holistic aggregate functions
In this research focus has been on the median aggregation function, which is a specific holistic aggregate function. It would be very interesting to investigate if the developed algorithms can also applied, likely in a modified way, towards other holistic aggregate functions. It is already possible to use this work for other n^{th} percentile functions, since the median is really just an alias for the 50th percentile, but investigation into the use of the work for other holistic functions, such as the top *k* aggregate function, would be most interesting.

Comparison with approximate algorithms
Several algorithms have been developed that are able to calculate the exact median of various data sets whilst trying to improve query performance to make real-time data exploration possible. Existing research into this field whereby the focus is on an exact median is very limited however much research has focused on employing algorithms that provide an approximate answer, as outlined earlier in section 2.2. It could be worthwhile to compare the test results of this research with test results of similar experiments with existing algorithms that provide an approximate answer to determine the cost, in terms of query performance and storage, of calculating an exact answer instead of an approximate answer.

The above five possible future research directions show the most promise and are likely to improve upon the foundation laid by this research into improving the query performance for holistic aggregate functions.

This section ends this thesis. On the next page you will find the bibliography, containing a list of all the references used in this thesis.

Bibliography

[1] Anastassia Ailarnaki, David DeWitt, Mark Hill, and David Wood. DBMSs on modern processors: Where does time go? VLDB, 1999.

[2] Daniel Barbar and Xintao Wu. Using approximations to scale exploratory data analysis in datacubes. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, page 382386, New York, NY, USA, 1999. ACM.

[3] Dina Bitton, David J DeWitt, and Carolyn Turbyfill. Benchmarking database systems: A systematic approach. In *VLDB*, volume 83, page 819, 1983.

[4] Peter A Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: hyper-pipelining query execution. In *CIDR*, volume 5, page 225237, 2005.

[5] F Braz, Salvatore Orlando, Renzo Orsini, A Raffaela, Alessandro Roncato, and Claudio Silvestri. Approximate aggregations in trajectory data warehouses. In *Data Engineering Workshop, 2007 IEEE 23rd International Conference on*, page 536545. IEEE, 2007.

[6] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. *The VLDB JournalThe International Journal on Very Large Data Bases*, 10(2-3):199223, 2001.

[7] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. A robust, optimization-based approach for approximate answering of aggregate queries. In *ACM SIGMOD Record*, volume 30, page 295306. ACM, 2001.

[8] Andy S Chiou and John C Sieg. Optimization for queries with holistic functions. In *Database Systems for Advanced Applications, 2001. Proceedings. Seventh International Conference on*, page 327334. IEEE, 2001.

[9] COMMIT. COMMIT, 2014.

[10] COMMIT. Community tagging | commit, 2014.

[11] Graham Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):5875, 2005.

[12] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining data streams: a review. *ACM Sigmod Record*, 34(2):1826, 2005.

[13] Phillip B Gibbons and Yossi Matias. New sampling-based summary statistics for improving approximate query answers. In *ACM SIGMOD Record*, volume 27, page 331342. ACM, 1998.

[14] Anna C Gilbert, Yannis Kotidis, S Muthukrishnan, and Martin Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *VLDB*, volume 1, page 7988, 2001.

[15] Anna C Gilbert, Yannis Kotidis, S Muthukrishnan, and Martin J Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *Proceedings of the 28th international conference on Very Large Data Bases*, page 454465. VLDB Endowment, 2002.

[16] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):2953, 1997.

[17] Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. In *ACM SIGMOD Record*, volume 30, page 5866. ACM, 2001.

[18] Stavros Harizopoulos, Velen Liang, Daniel J Abadi, and Samuel Madden. Performance tradeoffs in read-optimized databases. In *Proceedings of the 32nd international conference on Very large data bases*, page 487498. VLDB Endowment, 2006.

[19] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. *Range queries in OLAP data cubes*, volume 26. 1997.

[20] KNMI. Uurgegevens van het weer in nederland - download, 2014.

[21] Laks V. S. Lakshmanan, Jian Pei, and Jiawei Han. Quotient cube: How to summarize the semantics of a data cube. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, page 778789, Hong Kong, China, 2002. VLDB Endowment.

[22] Cuiping Li, Gao Cong, Anthony KH Tung, and Shan Wang. Incremental maintenance of quotient cube for median. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, page 226235. ACM, 2004.

[23] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th international conference on Very Large Data Bases*, page 346357. VLDB Endowment, 2002.

[24] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *ACM SIGMOD Record*, volume 27, page 426435. ACM, 1998.

[25] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. *ACM SIGMOD Record*, 28(2):251262, 1999.

[26] Microsoft. MSDN - PERCENTILE_DISC (transact-SQL), 2014.

[27] Arnab Nandi, Cong Yu, Philip Bohannon, and Raghu Ramakrishnan. Distributed cube materialization on holistic measures. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, page 183194. IEEE, 2011.

[28] Viswanath Poosala and Venkatesh Ganti. Fast approximate answers to aggregate queries on a data cube. In *Scientific and Statistical Database Management, 1999. Eleventh International Conference on*, page 2433. IEEE, 1999.

[29] Dean Rasheed. wip: functions median and percentile, 2010.

[30] Roger Schrag. Use EXPLAIN PLAN and TKPROF to tune your applications, November 2014.

[31] Tada AB. PL/Java, 2014.

[32] Jeffrey Scott Vitter, Min Wang, and Bala Iyer. Data cube approximation and histograms via wavelets. In *Proceedings of the seventh international conference on Information and knowledge management*, page 96104. ACM, 1998.