Efficient Aggregate Queries on Data Cubes

Fredrik Bengtsson

Division of Computer Science and Networking
Department of Computer Science and Electrical Engineering
Luleå University of Technology
Luleå
Sweden

Supervisor:

Jingsen Chen

Abstract

As computers are developing rapidly and become more available to the modern information society, the possibility and ability to handle large data sets in database applications increases. The demand for efficient algorithmic solutions to process huge amounts of information increases as the data sets become larger. In this thesis, we study the efficient implementation of aggregate operations on the data cube, a modern and flexible model for data warehouses. In particular, the problem of computing the k largest sum subsequences of a given sequence is investigated. An efficient algorithm for the problem is developed. Our algorithm is optimal for large values of the user-specified parameter k. Moreover, a fast in-place algorithm with good trade-off between update- and query-time, for the multidimensional orthogonal range sum problem, is presented. The problem studied is to compute the sum of the data over an orthogonal range in a multidimensional data cube. Furthermore, a fast algorithmic solution to the problem of maintaining a data structure for computing the k largest values in a requested orthogonal range of the data cube is also proposed.

Preface

My interest in computers and computer programming started fairly early, when I tried to learn how to program on a pocket calculator with Basic. However, I created miserable programs. Later I learnt Pascal and appreciated the power of the language compared to Basic.

However, it was not until I begun studying computer science at Luleå University of Technology I learnt how to program. During my years as an undergraduate student, I realized that I was not really interested in programming (later, I realized that I am, but that's another story), but in the methods and ideas that made the programs work. I learnt that those methods were called algorithms. I felt that algorithmic design was the essence of programming. I am very grateful for having the opportunity to discover such an interesting area.

I started as an undergraduate student at Luleå University of Technology in the fall of 1996 and became a Master of Science in 2001. The same year, I was accepted as a Ph.D. student at the Division of Computer Science.

The work presented in this thesis has been mostly fun, interesting, and entertaining, but sometimes also frustrating and difficult. However, most of all, it has just been a lot of work. I would like to thank my great supervisor Jingsen Chen for his tremendous engagement and support in every situation. Without him I don't know what I would have done. I would also like to thank all the people at the Division of Computer Science and Networking for helping me with all imaginable problems that have popped up from nowhere.

As computer science is a young research area, I think it will be very interesting to be able to follow, and even participate, in its development in the future.

Contents

Chapter 1
Introduction

Through the history, humans have always striven for knowledge; knowledge about anything and everything. The need for organizing and communicating newly gained knowledge and preserving important knowledge have always been a prime issue for the human mind. In todays information society, the means of gaining knowledge is larger than ever. Computers have had a large impact on applied science, but also on theoretical science. Computers have been available for scientists for slightly more than 50 years [Ran73, CKA96] and have already formed a research field of its own. Computer science and computational science is a young science, and the development of computers and computational methods have only begun to develop. It will be interesting to be able to follow the development in this field and its impact on other scientific research areas in the future.

In the next section, we will discuss some of the problems that arise when computers becomes more powerful. Efficient methods for storing and retrieve stored information will be discussed. We will introduce the concept of databases and the operations on them.

1.1 Data Structures and Algorithms

The development of modern computers have been very fast and there is no reason to believe that this will not be the case in the future. In fact, in terms of performance, the development speed is actually increasing [Moo65]. In order to be able to utilize the power of modern computers, there is an ever increasing need for fast and efficient data structures and algorithms [CLR90]. One may argue that modern computers are sufficiently powerful to operate fast without efficient algorithms. However, this is not the case, since the computational complexity of problems often increase much faster than the size of the problem. Modern computers with large memory systems allow very large problem instances. This tends to be problematic even for fast computers if not supported by efficient data structures and efficient algorithms. Additionally, the demands is increasing as new applications and new computations are investigated. In light of this, efficient algorithms and data structures will become more and more important as computers evolve.

1.1.1 Efficient and Simple Data Structures and Algorithms

As mentioned, efficient data structures and algorithms will be even more important as computers with larger memories becomes available. It is important to observe, however, that not only the time-related performance (time complexity) of data structures and algorithms is important, but also their *simplicity*. There are countless examples of algorithms for fundamental problems with efficiency not met by any other algorithm; still, these algorithms are never or seldom implemented and used in real applications. This is because of their structural and logical complexity. These algorithms are often very hard to implement. It is important, however, to emphasize that theses algorithms are in no sense useless. In fact, they often represent important advances in algorithmic design and are, therefore, basis for many other algorithms. Additionally, they are often optimal and as such, closes open research questions about optimal solvability of problems.

For practical applications, however, there is a need for efficient, yet simple, algorithms. There is no inherent contradiction between efficiency and simplicity. While there exists widely used measures for memory and time consumption, there is, unfortunately, no widely accepted measure for the simplicity of algorithms. Nevertheless, such measure will probably evolve as the importance of simple and efficient algorithms is made more clear.

Consider the problem of computing the sum of the elements in an array between to given indices. It may seem trivial, and it is; but is the trivial solution efficient? As the size of the array becomes larger, the time for the computation grows linearly with the size, in the worst case. It would be nice with a more efficient solution. If the prefix sums of the array is computed (Figure 1.1), we will be able to compute the sum of all elements between any indices in constant time (subtract the high index element from the low index element) (Figure 1.2). Now, updating the array requires linear time. The solution is as inefficient as the first. These two solutions represents the simplest possible algorithms, but their efficiency may not be sufficient. The obvious question is, do there exist more efficient algorithms that are still simple? The answer is yes and this thesis will present the answer to this and several other related questions.

The situation becomes even more obvious when considering a slightly harder problem: find the subsequence with the largest sum, that is maximize the range sum of the

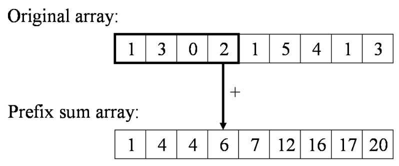

Figure 1.1: Example of the original array and the prefix sum. Element i in the prefix sum array is the sum of elements $1, 2, \ldots, i$ of the original array.

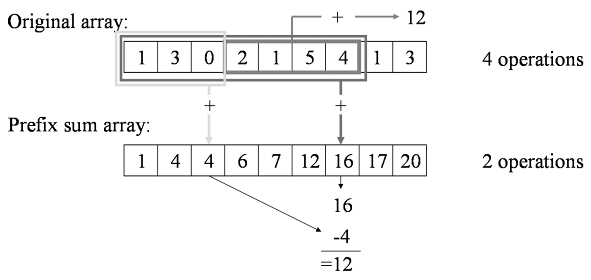

Figure 1.2: Example illustrating how to compute the range sum using the prefix sum instead of the original array. The computation of the range-sum with a value of 12 (the sum from element 4 to element 7) is performed by subtracting element 3 of the prefix sum from element 7 of the prefix sum

previous problem. If we not only want to compute the largest, but the k largest sums, the performance will be even worse. This is one of the problems that we consider in this thesis. The solution is simple enough to be implemented easily.

1.1.2 Fast Processing of Databases

A large amount of the computers that are in use today is a part of the global Internet. The massive amounts of information present in such a huge network is overwhelming. The Internet is one good example of a huge distributed database. Search engines accommodate a large fraction of the information on the Internet and makes it available for searching. There are, of course, a variety of scientific applications that require fast access to large amounts of data. Modern radio telescopes and VLBI (Very Long Base Line Interferometry) arrays collects huge amounts of data under relatively short time. If this data can be processed in real time there is a large benefit from a scientific point of view.

Modern computers have massive amounts of memory, which enables the storage of very large data sets. Often, as the size of the data set increases, the time required to perform operations on such data sets increase even faster. For instance, if the size of the data set is doubled, the time required to perform such an operations might be quadrupled. Because of this, the need for fast and efficient algorithms with low time complexity becomes more important as computer memory becomes larger. Faster processors does not compensate for the performance loss. This thesis presents several algorithms for aggregate information processing that have better performance than the previously best known algorithms. In the next section, we will introduce the models for databases and operations related to them.

1.2 Databases and Their Operations

Large databases have always been an issue for powerful computers. It is an obvious advantage to use computers in this area and, therefore, databases was one of the first large applications of computers [CKA96]. There is, of course, several reasons for the popularity of computers in these specific areas. One of the most important motivation for the use of computers in those areas are the enormous time needed to perform the corresponding operations by hand. Often, the time required to perform database and (more so) scientific computing without computers made the problems intractable.

1.2.1 Database Models

The widespread use and acceptance of computer-based systems for database management have lead to a variety of different conceptual models for database management and data view.

One of the early models that appeared in the context of database management systems is the *network data model* [cod71, EN00]. The model was later revised [cod78]. In this model, data is stored in *records*, where each record consists of several *fields*. Each record has a *record type* which specifies the fields of that particular record type and the types of the fields. Records are associated into *sets* which, in this context, is an 1-to-n relationship: The set has a *set owner* from one record type and one or several *set members* from another record type. The sets describe the relations between records. See Mary E. S. Loomis [Loo80] for a survey.

Another early database model that has gained a lot of interest is the *hierarchical data model* [EN00]. In this model, the data consists of records, in the same way as in the network data model, but the relations among records are defined by a parent-child relationship in a tree-like manner. Records are called *nodes* in this context. The system is considered parent of the root of the tree and nodes that are not parent of any other node are termed *leaf nodes*.

In many of today's database management systems, a database models with thorough mathematical foundation is used: the *relational database* [Cod70, Cod95, EN04]. The relational database have gained a lot of interest because of its flexibility and mathematical foundation. The relational model is based on the mathematical concept of relational algebra. In this model, the data is, as in many other models, represented by *records* that consists of *fields*. Both records and fields have a *type* that specifies the type of informational content. The relational database model introduces the concept of *tables* of records. Each record present in the database of a certain type can be organized in a table where each row corresponds to a record and each column corresponds to a field. The database can consist of many tables of different record types. Each such table defines a *relation* on the record type. Each record can be viewed as specifying a point in a multidimensional universe (one field for each dimension) and hence the term relation.

Alongside the database models there are two common ways of characterize the use of the database and its purpose. This can impact the way data are handled by the database management system in the sense that query types have different frequency in different models and the amount of data required for a query varies between the models. The traditional view that is often used by banks, insurance companies and governments is the *On-Line Transactional Processing* (OLTP). It refers to the way the database is accessed and maintained. In OLTP, the flow of new information into the database is potentially large and each event in a system (an update, for instance) is called a *transaction*. The characteristics for such a system is the ability to handle a large amount of transactions per time unit in order to provide users with a fast database for both lookups and updates. Notice that each single transaction is not a very complex operation. It either requests some information (of fairly basic type) from the database or it updates the database.

Lately, the *On-Line Analytic Processing* (OLAP) [BE97] has gained interest. In this model, the focus is on aggregating the information in the database in order for the user to get a compiled view of some part of the data in the database. This kind of database is often used for in a decision process by an organization that needs to make decisions based on the combined content of the database. The amount of data required to be processed by a query can be huge, potentially the whole database. On the other hand, it may not be as important as in the OLTP model to have access to the most recent data. This can often have a dramatic performance impact on the database and be in conflict with the requirement of the OLTP queries and updates. Therefore, the data is often moved from the OLTP system to a separate database, called *data warehouse* that is specialized in answering OLAP queries fast. It is of great importance to be able to achieve high performance for OLAP queries as well, not just OLTP. This is because the queries often repeatedly needs to be reformulated and the database required in order to fit the needs of the user and give the user a complete view of the data of interest.

1.2.2 The Data Cube

In the context of OLAP and data warehouse the *datacube* is often used as a concrete view of the database. The datacube can be regarded as a different view of a relation from a relational database. One of the fields (often called *attributes* in this context) from the relational database's record is chosen as the *measure attribute* and the other are called *dimensional attributes*. The origin of the name "datacube" is from the view of the set of records as a multidimensional array (a cube, in three dimensions), where the dimensional attributes are used as orthogonal axes and the measure attribute as a value in the *cell* for each record. Notice that there is no real difference between the measure attribute and the dimensional attributes (there can, of course, be a semantic difference). It is possible to use all attributes as dimensional attributes and merely denote the presence of a record by a single bit in the datacube. However, the distinction between dimensional attributes and measure attributes are convention. This is probably because the datacube can easily be implemented in this way with a multidimensional array.

Operations on the Datacube

Since the OLAP model is targeted at analysing data already present in the database in an aggregate and overview fashion, the supported operations on the datacube implementation of the OLAP model typically incorporates information from a huge number of elements into the answer of the query. The need for high performance in the answering of advanced aggregate operations sometimes leads to algorithms with poor update performance or that does not at all allow updates. This can be acceptable in some instances of OLAP systems. However, in many systems there is a need for fast updates as well. High update performance of aggregate operations is, for many query types, an important research topic. We will show some progress in this thesis.

The range max/min query is one of the basic datacube operations. The query consists of a multidimensional range specified as ranges over the dimensional attributes of the datacube. The answer is the maximum (or minimum) value over all records (or, points) in the query region. The range max/min query only operates with orthogonal ranges.

A similar and often used query is to retrieve, not only the largest value in the range, but also the second largest, the third largest, and so on, from the range. The *range top-k query* [LLAL02, LLA$^+$01] or *k max*, retrieves the largest, the second largest, and so on, the kth largest elements of a specified range. With such a query, more information is retrieved from the datacube and the user is provided with a better overview of the information.

It is often also of interest to perform mean and median operations on selected parts of the dataset represented by the datacube. The range mean and range median [KMS03] queries are two query types that can be efficiently supported, on static datasets. The range is specified in the same manner as in the range max/min query.

A very powerful aggregate operation on the datacube is the ability to identify a region which has the largest sum of values. This leads to the multidimensional version of the classical maximum sum subsequence problem, where the region of the largest sum of the datacube is to be found. A more general operation is the *k maximum sum subsequences* problem. It is the problem to finding the *k* largest range sums.

The user demands on the data warehouse and on the OLAP environments to perform the abovementioned operations well have lead to the intensive study of corresponding algorithmic problems. These fundamental problems in computer science are of great interest, not only in the context of data warehousing and OLAP, but also in other areas, including machine learning [Maa94], computer graphics [Gre78] and computational biology [mCL03].

In the next section, we will describe the research that has been performed on these problems and present our new results.

1.3 Algorithmic Problems

Much effort has been spent in order to efficiently support query operatins in large datasets, such as aggregate queries and range queries, in the OLAP environment. In this section, some of the underlying algorithmic problems that arise in the context of databases and data warehousing are described and the previous results are presented. Our new algorithms for the problems and their performance will be introduced briefly as well.

1.3.1 Orthogonal Range Sum

There are a number of efficiency parameters to consider when designing algorithms for the range-sum query, including space usage, update performance, query time and the ability to support other operations than just the range-sum. Ho et al. [HAMS97] introduced the prefix sum approach for the solution. With this approach, the query performance is $O(2^d)$, independent of the size of the query region, where d is the dimensionality of the data cube. The space usage is optimal: It requires the same amount of space as the data cube itself. The update performance, however, is $O(n^d)$ for data cubes of n^d elements. In the same paper, Ho et al. also suggest using a block version of the prefix sum technique, which improves the update performance, but increases the space usage. Moreover, the query performance also decreases accordingly. Additionally, Ho et al. also in the same paper, suggest batching updates and perform several updates at one instance and thereby saving time per query when rebuilding the prefix cube. Along this line several other techniques, based on the prefix sum approach, have successfully been developed [GAAS99, CCLL01, GRAA99, RAAP00, RAA01, Poo01].

Poon [Poo01] introduces a data structure with a query time of $O((2L)^d)$ and update time of $O((2Ln^{(1/L)})^d)$, where L is a parameter. The structure uses $O((2n)^d)$ storage. Poon also proposes efficient similar structures for the range max problem. Geffner et al. [GAAS99, GRAA99] introduce an efficient structure, termed the *Relative Prefix Sum* that has a query complexity of $O(2^d)$ and an update complexity of $O(n^{d/2})$. The space usage of this structure is also super-linear, but smaller than that of the structure by Poon. Geffner et al. [GRAA99] also proposes a recursive structure, termed the *Dynamic Data Cube*, which has update and query complexity of $O(\log^d n)$. Again, the space usage is super-linear but less than the space usage in Poon's structure.

Chun et al. [CCLL01] present a structure with good performance according to their simulations. However, the space usage is still super-linear.

The above-mentioned structures uses extra space in order to achieve high performance for queries or updates or, both. Two structures saving space have brrn introduced by Riedewald et al. [RAAP00]. One of them is the *Space Efficient Relative Prefix Sum* which is a direct successor to the *Relative Prefix Sum*. It does not use any extra space, but has the same performance as the Relative Prefix Sum, which is $O(n^{d/2})$ for updates and $O(2^d)$ for queries. The other structure that is the *Space Efficient Dynamic Data Cube* by Riedewald et al. [RAAP00] that has the same performance as its predecessor, the Dynamic Data Cube, namely $O(\log^d n)$ for both updates and queries. This query performance is, however, for prefix sum queries.

An interesting special case is when the data cube is considered to be sparse, and thus, the space savings could be huge if the sparsity is taken into account. This have been studied by Chun et al. [CCLL02]. They propose a technique called *PC-pool* for sparse cubes. They demonstrate the performance of the PC-pool by simulations.

New Results

In Chapter 2 we show how to answer range-sum queries on the data cube with constant query complexity while maintaining sub-linear update time. Our algorithm does not use any extra space. These results provides superior query-update trade-off over previous known structures. The structure should also be fairly easy to implement. The performance of our structure is the result of further development of both the Space Efficient Relative Prefix Sum and the Space Efficient Dynamic Data Cube.

In addition to both better update and query performance, our structure has a trade-off parameter, which can be used to trade query performance for update performance. The query performance can be held constant, with respect to the size of the data cube, while the update performance is highly super-linear. The query performance of our structure is $O\left(2^{id}\right)$ while maintaining an update performance of $O\left((2^i \sqrt[2^i]{n})^d\right)$, where i is the trade-off parameter. Theses results have previously been published in *Lecture Notes in Computer Science* [BC04b].

1.3.2 k Maximum Sum Subsequences

The maximum sum subsequence problem is to find a contiguous subsequence of elements in a sequence that has the largest sum of all contiguous subsequences. The problem was first introduced by Bentley in 1984 [Ben85b, Ben85a]. In one dimension, there exists a linear time sequential solution [Ben85b, Gri82]. In the multi-dimensional case, however, no optimal solution has been developed. For an $m \times n$ matrix of real numbers, the maximum sum subsequence problem can be solved in $O\left(m^2 n\right)$ time (assuming that $m \leq n$) [Ben85b, Gri82, Smi87]. By reducing the problem to graph distance matrix multiplications, Tamaki and Tokuyama [TT98] present the first sub-cubic time algorithm for the two-dimensional maximum sum subsequence problem. The time complexity of their algorithm is

$$O\left(nm^2\sqrt{(\log\log m/\log m)}\log(n/m)\right) .$$

They also present an ϵ-approximation algorithm with a time complexity of

$$O\left(\epsilon^{-1}(-\log\epsilon)nm^{\omega-1}\log(n/m)\right) .$$

Here, ω is the exponent in the complexity of matrix multiplication; Currently, $\omega = 2.376$ [CW90]. Following the same strategy, Takaoka [Tak02] gives a modified algorithm with a slightly better time complexity, namely

$$O\left(nm^2\sqrt{(\log\log m/\log m)}\right) .$$

In the context of parallel computations, optimal speed-up algorithms on different models of parallel computations have been developed [Smi87, AG91, PD95, QA99].

A natural extension of the above problem is to compute the k largest sums over all possible subsequences/subarrays instead of just the largest. Recently, a $\Theta(nk)$-time algorithmfor this problem has been presented [BT04].

New Results

In Chapter 3, we will present an algorithm that solves the k maximum subsequences problem in $O\left(\min\{k+n\log^2 n, n\sqrt{k}\}\right)$ time, in the worst case. This is optimal for $k \geq n\log^2 n$ and improves over the previously best known result for any value of the user-defined parameter k. Moreover, our results are also extended to the multi-dimensional versions of the k maximum sum subsequences problem; resulting in fast algorithms as well. These results will appear in *Lecture Notes in Computer Science* [BC04a].

1.3.3 Top-k Queries

In Chapter 4, we will study the *Range top-k* problem. The problem is to find the k largest values in some specific range and is a generalization of the range max problem. The problem has been studied by Donjerkovic and Ramakrishnan [DR99]. They use a probabilistic approach to solve the problem. Chaudhuri and Gravano [CG99] studies a variant of the problem, where the objective is to find the largest k records that are located nearest to the query point. They present their algorithm in the context of the relational database model. Luo et al. [LLA$^+$01] studied the range top-k query in the context of sparse data cubes. For the general case Loh et al. [LLAL02] introduce the *Adaptive Pre-computed Partition Top method* (APPT) to solve this problem. APPT performs well as shown by experiments [LLAL02]. We will show, in Chapter 4 of this thesis that the worst-case query complexity of their algorithm is $O(n^d)$ for a d-dimensional data cube.

Moreover, we design an $O(m_1+m_2)$-query algorithm for the two-dimensional case, where $m_1 \times m_2$ is the size of the query region. Our structure is similar to the APPT in that blocks are used to store pre-computed values. However, the new structure is recursive in hierarhcical levels.

Chapter 2

Space-Efficient Range-Sum

In this chapter, we study the problem of range-sum on a multidimensional data cube. The problem is to find the orthogonal range-sum of a data cube and has been investigated intensively. Previously, two space-efficient structures, the *Space Efficient Dynamic Data Cube* [RAAP00] and the *Space-Efficient Relative Prefix Sum* [RAAP00] achieve good performance. We will generalize theses two results and propose a new data structure and algorithm for the problem. Our new algorithm has better update-query time trade-off than previous algorithms. As a special case, our algorithm can guarantee constant query time while maintaining sub-linear update time.

Moreover, we will analyze the complexity of the Space Efficient Dynamic Data Cube which are omitted in the original paper [RAAP00].

2.1 Prefix Sum Methods

This section will review some previous methods that are closely related to our new structure. All those structures are based on the prefix sum [HAMS97]. Among them are the Space-Efficient Relative Prefix Sum [RAAP00] and the Space-Efficient Dynamic Data Cube [RAAP00].

Consider the two extremes: the original array of length n and the prefix sum array. A query performed on the original array takes $\Theta(n)$ time in the worst case. An update, on the other hand, requires only $O(1)$ time. For an array of prefix sums, it's the opposite; an update requires $\Theta(n)$ time (because the prefix sum needs to be

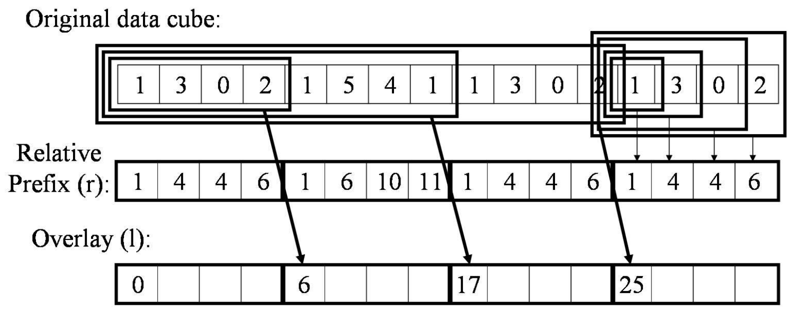

Figure 2.1: One dimensional Relative Prefix Sum

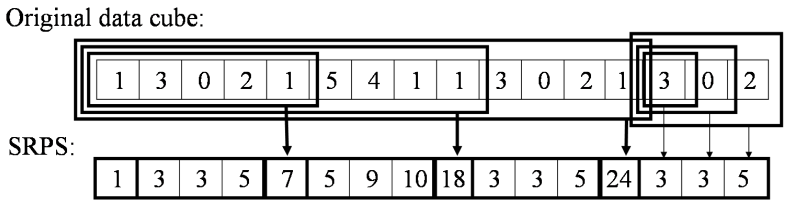

Figure 2.2: One dimensional Space-Efficient relative Prefix Sum

recalculated), while a query requires only $O(1)$ time. Different techniques have been proposed that combines these two techniques.

The Relative Prefix Sum [GAAS99, GRAA99] and the Space-Efficient Relative Prefix Sum [RAAP00, RAA01] use a clever blocking scheme to achieve their high performance. Each dimension of the data cube is divided into $\sqrt{n}$ blocks. Consider a one-dimensional data cube (an array). Basically, for each block, a prefix sum of all elements in the block is computed. Additionally, for each block, the sum of all elements of previous blocks is stored. This makes it possible to compute the prefix sum by adding one element from the prefix sum of the block and the other element (the sum of all previous blocks) corresponding to the same block. Updates will now be faster, since only the prefix sum in one block need to be updated. Additionally, all of the other elements has to be updated, but that requires the same amount of time as the update of the prefix sum.

To be precise, in the Relative Prefix Sum (RPS) [GAAS99, GRAA99], the data set is represented by two arrays; the *Overlay* array (l) and the *Relative prefix* array (r). Each array is divided in $\sqrt{n_i}$ blocks of $\sqrt{n_i}$ elements each, along dimension i, for $i = 0, 1, \ldots, d-1$. In Figure 2.1, a simple one dimensional example of the RPS is shown, where $n = 16$ and the block boundaries have been emphasized with thick

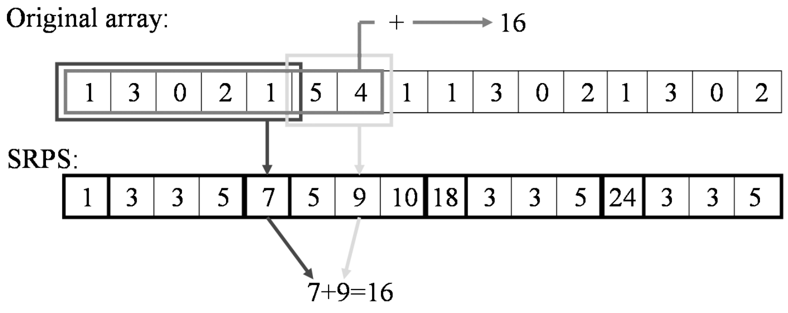

Figure 2.3: Computing the prefix sum using the Space-Efficient relative Prefix Sum. The two elements from the SRPS is added in order to form the prefix sum.

lines. In the relative prefix array, r, the prefix sum relative to each block is stored. In the overlay array, l, the first element of each block stores the sum of all elements (from the data cube) of all previous blocks. In d dimensions, the situation is slightly more complicated, but the idea is the same. A query is performed by adding the value from the block of the overlay array, corresponding to the higher index of the query range, with the value from the relative prefix array with the higher index of the query. This effectively computes the prefix sum for the upper query index. For example, consider Figure 2.3. Here, the prefix sum for index 7 is computed using the SRPS. Clearly, the prefix sum is 16 (the sum of all elements from the first up to the 7th). Using the SRPS, the sum is computed by adding the element at position 5, which has the value 7, with the element at position 7, which has a value of 9. The sum, $7 + 9$, is 16. For a range-sum query, the prefix sum for the lower query index is computed in the same way and the two are subtracted to give the result. When updating the structure, it is only necessary to update the block of the relative prefix array corresponding to the updated element. There are $O(\sqrt{n})$ elements in this block. The overlay array may need to be rebuilt. The overlay array consists of $O(\sqrt{n})$ elements, so the total update complexity is $O(\sqrt{n})$.

The Space-Efficient Relative Prefix Sum (SRPS) by Riedewald et al. [RAAP00] is a structure very similar to the RPS, but the space usage is smaller. Consider the one-dimensional RPS of Figure 2.1 again. If the elements in the overlay array contains the inclusive sum (instead of the exclusive), we need only store a shorter local prefix sum in the RPS array. It is sufficient that the local prefix sum in the RPS array starts at the 2:nd position in the block (if the overlay sums are inclusive). The first position in the local prefix sum in the RPS array can then be used to store the overlay array. This way, the structure does not use more space than the original data cube (Figure 2.2). Search and update are performed similar to the RPS.

In the Space-Efficient Dynamic Data Cube [RAAP00], which is an improvement of the *Dynamic Data Cube* (DDC) [GRAA99], the technique of the SRPS is applied recursively on each block and on the boundary cells of each block (this only makes sense in two or more dimensions). However, instead of splitting the array into $\sqrt{n_i}$ blocks, each dimension is split in two (or any other constant) parts. In general, we get 2^d boxes for a d-dimensional cube. Now, call the region corresponding to the prefix sum of a block, the *inner region* of each block. The inner region, in d dimensions, contains all elements in a box, except for the first element in *each dimension.* Those elements form the *border region.*

The inner region is a prefix sum and it is possible to recursively apply the algorithm to this region in the following manner. Consider the data cube that would have generated the prefix sum (the pairwise difference of the prefix sum). It is possible to recursively apply the SDDC algorithm to that data. Note that it is no longer necessary to store the original elements of the inner region, but the elements resulting from the recursive application of the algorithm is stored instead. The recursion continues $O(\log n)$ levels.

Now, consider the border cells of a box (cells that are not in the inner region – the first cell of each block, in the one-dimensional case). Observe that, in the one-dimensional case, the border of each block has dimension 0 (it is a scalar). In general, for d dimensions, the border cells contains $\binom{d}{k}$ regions of k dimensions. For instance, in two dimensions, there are two regions of one dimension and one region of zero dimensions. Each border region is in itself a prefix sum because each element (border cell) is the sum of all elements in the data cube with smaller index. Each such sum span $d-k$ dimensions. For each region of the border of each box, the SDDC algorithm (of the dimensionality of the region) is applied recursively. The elements are stored in the same place as we would have done without recursion, thus, throwing away the old elements (from before the recursion). Queries and updates are handled similar to the SRPS, but recursively.

This way, we get a structure (and a corresponding algorithm) requiring $O(\log^d n)$ array accesses for both update and query. See Section 2.3 for a performance analysis, which is omitted in the original paper [RAAP00].

2.2 Our Result

In our structure, the recursive technique of the DDC and the SDDC is extended significantly. Our structure uses the same blocking scheme as the SRPS structure, but the algorithm is not only applied recursively within each block. This would indeed yield worse performance than the SDDC. Instead, we observe that the extra element stored for each block together with the extra elements from all other blocks, form a prefix sum of their own! It is then very possible to recursively apply the algorithm to those elements in addition to the blocks. The real benefit of this approach is seen only in higher dimensions, however. Here, the DDC and the SDDC have to use recursive calls to the algorithm of lower dimensions, in fact all dimensions lower than the actual data cube. This complicates the analysis quite a bit. Nevertheless, an analysis of the SDDC is provided in Section 2.3. In our algorithm, all recursive subproblems will be of the same dimensionality as the original data cube, which makes the structure very easy to analyze in addition to giving it high performance.

Our structure for range sum queries combines the techniques from both the SRPS and the SDDC. We use a recursive storage technique to obtain fast range sum queries (constant time) and to achieve good trade-off between queries and updates. In essence, instead of storing the elements from smaller subproblems that appear in the structure directly, our algorithm is applied recursively and the result from the recursion is stored. This results in a query performance of $O(2^{id})$ while maintaining an update complexity of $O((2^i \sqrt[2^i]{n})^d)$, where i is a trade-off parameter. This improves over previous known structures, in the asymptotic sense. The trade-off can be tuned toward query performance and achieves constant query time while maintaining a highly sub linear update complexity. Observe that the complexity of updates can be improved by just increasing the time consumed by queries. Notice, however, that it is possible to improve the complexity of updates while still maintaining constant query time.

2.2.1 Constant Query Time

The query time of $O\left(2^{id}\right)$ is constant with respect to n for any i and d. Thus, it is possible to achieve very good update performance while still maintaining constant lookup time. We will first present the general idea and then continue with a detailed presentation of the two dimensional case followed by the general case.

The algorithm splits each dimension in $\sqrt{n}$ blocks (each with side-length $\sqrt{n}$). Consider Figure 2.5. We make the important observation that it is not only possible to apply the algorithm recursively *within* each block, but that all border regions of k dimensions together with the corresponding border regions from other cells in $d-k$ dimensions form a subproblem of size $O(\sqrt{n})$ and dimensionality d. See Figure 2.5 for a 2-dimensional example of the recursion technique. Here, one row of 1-dimensional subproblems (a) form a new 2-dimensional subproblem. In the same way, one column of 1-dimensional subproblems (b) form a new 2-dimensional subproblem. Additionally, the first element of all blocks (0-dim) form a new 2-dimensional subproblem (c). Thus, we only have to recurse over subproblems of the same dimensionality. In this way, we can not only describe our fast algorithm simply, but perform the complexity analysis of the algorithm neatly.

Consider the one-dimensional example of Figure 2.2 again. At just one recursion level, our structure would be the same as the SRPS. However, at two recursion levels, the structure would be formed in each of the inner regions as well (in order from left: $\langle 3,3,5\rangle$, $\langle 5,9,10\rangle$, $\langle 3,3,5\rangle$ and $\langle 3,3,5\rangle$). Unfortunately, it takes a much larger example to actually form a two-level structure. Each border region (in order from left: $\langle 1\rangle$, $\langle 7\rangle$, $\langle 27\rangle$ and $\langle 32\rangle$) together form a subproblem. The structure is recursively applied to them as well. This would yield the vector $\langle 1,6,27,5\rangle$. This vector would

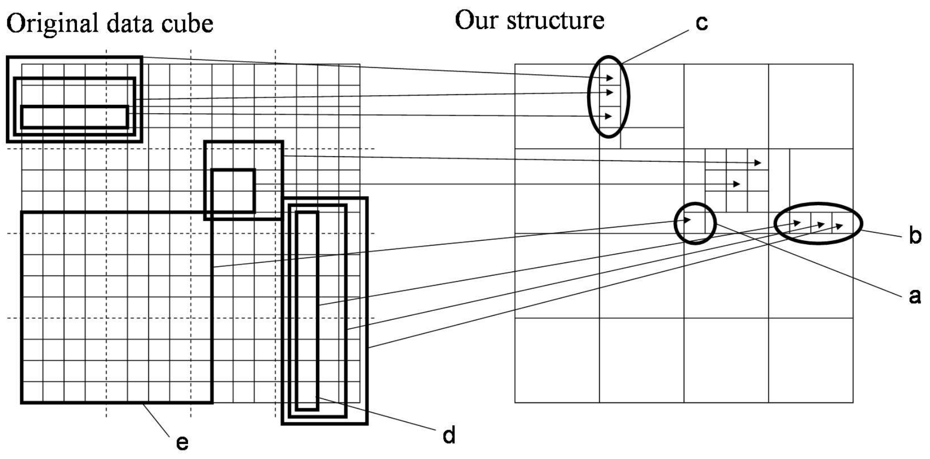

Figure 2.4: 2-dimensional example

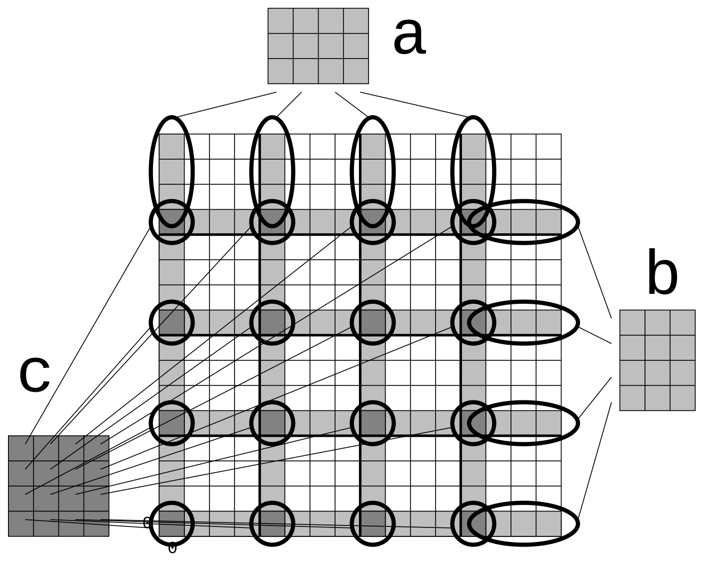

Figure 2.5: Recursion in a 2 dimensions

then be stored instead of the border elements. The structure is applied to the pairwise difference of the elements ($\langle 1, 6, 20, 5\rangle$), not the elements themselves. The inner regions in this very small example becomes only one element (the 6 and the 5).

Let d be the dimension of the data cube and $n_i, 0 \leq i \leq d-1$ be the size of the data cube for dimension i. Moreover, let $e_{(x_0,x_1,\ldots,x_{d-1})}$ denote the elements of the original data cube, where $0 \leq x_i \leq n_i - 1$. and let $g_{(x_0,x_1,\ldots,x_{d-1})}$ denote the elements of our data structure. Next, we will present the data structure in detail. First, the two-dimensional case is presented, where after the general case is introduced.

Consider a single block

$$b_{(x,y)} = g_{(k_0\lceil\sqrt{n_0}\rceil+x,k_1\lceil\sqrt{n_1}\rceil+y)} \ ,$$

where $x \in \left[0, \left\lceil\sqrt{n_o}\right\rceil - 1\right]$, and $y \in \left[0, \left\lceil\sqrt{n_1}\right\rceil - 1\right]$ for any k_0 and k_1 (such that the block is within the data cube) of an n_0 by n_1 structure of two dimensions. Then $b_{(0,0)}$ (marked "a" in Figure 2.4) will contain the sum of all elements of the original data cube that has smaller index than both x and y (the elements to the lower left, marked "e", in Figure 2.4). Next, consider the elements

$$\left\{b_{(x,0)} : x \in [1, \lceil\sqrt{n}\rceil - 1]\right\}$$

(marked "b" in Figure 2.4). The element $b_{(1,0)}$ will contain the sum of all elements from the original data cube with smaller y-index than $b_{(1,0)}$, but with the same x-index. This is shown with "d" in Figure 2.4. Observe that it is not the elements of the structure that is summed, but the elements of the original data cube. In the same way, $b_{(2,0)}$ contains the sum of the elements with smaller y-index, but with $x = 2$ plus $b_{(1,0)}$. This holds in general. The fact that

$$b_{(x,0)} = b_{(x-1,0)} + e_{(k_0\lceil\sqrt{n_0}\rceil+x,k_1\lceil\sqrt{n_1}\rceil)}$$

makes the elements a prefix sum. The set of elements

$$\left\{b_{(0,y)} : x \in [1, \lceil\sqrt{n_0}\rceil - 1]\right\}$$

(marked "c" in Figure 2.4) stores elements in the same way, but with coordinates swapped.

We observe that each block in the two-dimensional case contains one prefix sum of two dimensions (white in Figure 2.5), two prefix sums of one dimension (light grey in Figure 2.5) and one prefix sum of zero dimension (a scalar, dark grey in Fig. 2.5). In general, for d dimensions, each block will contain $\binom{d}{k}$ prefix sums of k dimensions. This can be realized from an algebraic point of view. The elements for the prefix sums within a single block can be obtained by fixing selected dimensions to 0 (relative to the block) and letting the other dimensions vary to span the region of the prefix sum. If we choose to fix k of the dimensions, it can be done in $\binom{d}{k}$ ways.

Consider an arbitrary block

$$b_{(i_0,i_1,\ldots,i_{d-1})} = g_{\left(k_0\lceil\sqrt{n_0}\rceil+i_0,k_1\lceil\sqrt{n_1}\rceil+i_1,\ldots,k_1\left\lceil\sqrt{n_{d-1}}\right\rceil+i_{d-1}\right)}$$

of our structure, where $k_0, k_1, \ldots, k_{d-1}$ selects block. Then we have that

$$b_{(i_0,i_1,\ldots,i_{d-1})} = \sum_{j_0=l_0}^{h_0} \sum_{j_1=l_1}^{h_1} \cdots \sum_{j_{d-1}=l_{d-1}}^{h_{d-1}} e_{(j_0,j_1,\ldots,j_{d-1})}$$

where

$$i_j = 0 \Leftrightarrow l_j = 0, h_j = k_j \left\lceil \sqrt{n_j} \right\rceil$$

and

$$i_j > 0 \Leftrightarrow l_j = k_j \left\lceil \sqrt{n_j} \right\rceil + 1, h_j = k_j \left\lceil \sqrt{n_j} \right\rceil + i_j$$

for

$$j \in [0, d-1]$$

iff

$$\exists i \in \{i_o, i_1, \ldots, i_{d-1}\} : i = 0 \ .$$

The last condition restricts the validity of the formula to the borders of the block (where at least one index is zero). For the rest of the block (the local prefix sum, no indices are zero) we have that

$$b_{(i_0,i_1,\ldots,i_{d-1})} = \sum_{j_0=a_0+1}^{a_0+i_0} \sum_{j_1=a_1+1}^{a_1+i_1} \cdots \sum_{j_{d-1}=a_{d-1}+1}^{a_{d-1}+i_{d-1}} e_{(j_0,j_1,\ldots,j_{d-1})}$$

for

$$i_j \in [1, \left\lceil \sqrt{n_j} \right\rceil - 1], a_j = k_j \left\lceil \sqrt{n_j} \right\rceil, j \in [0, d-1] \ .$$

The above discussed structure is a one-level recursive structure (that is, $i = 1$).

Now, we are ready to present the recursion for the general case. Consider Figure 2.5. Within each block (the inner region), the algorithm can be applied recursively. Consider the prefix sum (white elements) of the inner region of a block. The algorithm could be applied to the original data cube that corresponds to the prefix sum (the pairwise difference). Consider element $(0,0)$ of *all* blocks (the dark grey elements). They also represent a prefix sum and our algorithm can be applied recursively to their pairwise difference. To see this, remember that each of those elements contains the sum of all elements from $e_{(0,0)}$ up to $b_{(0,0)}$ (in two dimensions). Thus, element $(0,0)$ from *all* blocks together forms a 2-dimensional prefix sum. Similar reasoning can be applied to elements

$$\{b_{(x,0)} : x \in [1, \lceil \sqrt{n_0} \rceil]\} \text{ and } \{b_{(0,y)} : y \in [1, \lceil \sqrt{n_1} \rceil]\}$$

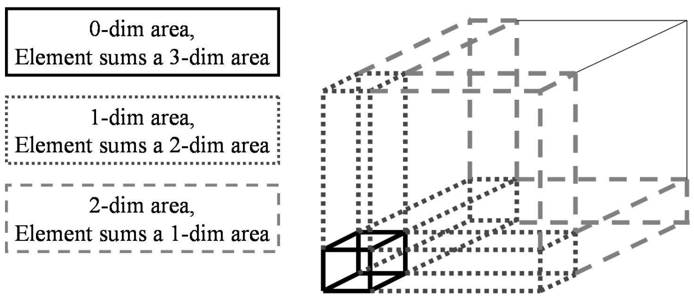
0-dim area,
Element sums a 3-dim area
1-dim area,
Element sums a 2-dim area
2-dim area,
Element sums a 1-dim area

Figure 2.6: Single box of 3-dimensional structure

(light grey in Figure 2.4). One row (or one column) of such elements form a two dimensional prefix sum and our algorithm can be applied recursively to those sums.

Instead of storing the elements of the structure directly, we store the elements of the recursive application of the algorithm. This is possible since the structure does not require extra space, and therefore, a recursive application does not increase the space requirement. Observe that the size of each dimension of the subproblems is $O\left(\sqrt{n_i}\right)$. In general, the subproblems consists of the elements from the data cube that can be obtained by varying the index *within* blocks for certain dimensions and varying *the block* for the other dimensions. As an example, we show a single box for our structure in 3 dimensions in Figure 2.6.

For an update, on a one-level structure, it is necessary to rebuild one block and all of the affected border regions (regions that include the updated element in their sum). However, for a multi-level recursive structure, it is only necessary to update the smaller block in the recursive application and the smaller border regions. A query can be processed fairly simple: The prefix sum (or the range-sum) can be computed by following the recursive structure and adding the appropriate elements.

With the above compact recursive structure, our algorithm achieves the constant time range sum queries.

Theorem 1. *The algorithm, described above, has a range-sum query complexity of $O(2^{id})$, where i is the recursion depth and d is the dimension of the data cube.*

Proof. Let $T_d(n,i)$ denote the query complexity. Clearly,

$$T_d(n,i) = \begin{cases} \sum_{j=0}^{d} \binom{d}{j} T_d\left(\sqrt{n}, i-1\right) & \text{, if } i > 0 \\ 1 & \text{, if } i = 0 \end{cases} .$$

For $i > 0$, we have

$$\begin{aligned}
T_d(n,i) &= \sum_{k=0}^{d} \binom{d}{k} T_d\left(\sqrt{n}, i-1\right) \\
&= 2^d T_d\left(\sqrt{n}, i-2\right) \\
&= 2^{2d} T_d\left(\sqrt[2^2]{n}, i-3\right) \\
&\vdots \\
&= 2^{jd} T_d\left(\sqrt[2^j]{n}, i-j\right) \\
&\left[\begin{array}{l} T_d(n,0) = 1 \\ i-j=0 \Leftrightarrow i=j \end{array}\right] \\
&= 2^{id} T_d\left(\sqrt{n}, i-i\right) \\
&= 2^{id} \quad .
\end{aligned}$$

□

At the same time, our algorithm requires sub-linear time for update operations while maintaining constant range-sum queries.

Theorem 2. *The algorithm, described above, has an update complexity of $O(2^{di}\left(\sqrt[2^i]{n}\right)^d)$, where i is the recursion depth and d is the dimension of the data cube.*

Proof. Let $U_d(n,i)$ denote the update time. Hence, from the description of the algorithm we have,

$$U_d(n,i) = 2U_d(\sqrt{n}, i-1), \text{if } i > 0 \quad ,$$

and

$$U_d(n,i) = n^d, \text{if } i = 0 \quad .$$

For $i > 0$, we have

$$\begin{aligned}
U_d(n,i) &= \sum_{k=0}^{d} \binom{d}{k} U_d\left(\sqrt{n}, i-1\right) \\
&= 2^d U_d\left(\sqrt{n}, i-2\right) \\
&= 2^{2d} U_d\left(\sqrt[2^2]{n}, i-3\right) \\
&\vdots \\
&= 2^{jd} U_d\left(\sqrt[2^j]{n}, i-j\right) \\
&\quad \left[\begin{array}{l} U_d(n,0) = n^d \\ i-j=0 \Leftrightarrow i=j \end{array}\right] \\
&= 2^{id} U_d\left(\sqrt{n}, i-i\right) \\
&= 2^{id}\left(\sqrt[2^i]{n}\right)^d \quad .
\end{aligned}$$

□

2.2.2 Update-Query Time Trade-Off

Notice that the time complexity of range sum query and of update depends not only on the data size n^d, but also on the number of recursion levels, i, of the data structure (parameter). By choosing the parameter i, we obtain different trade-off between the running time of queries and updates. Hence, i can be chosen to tailor the performance of the structure to the needs of the user. If queries are much more common than updates, probably a constant query time is desirable. If $i = O(1)$, then the query complexity is always constant (with respect to n), but the update complexity is

$$O\left(\sqrt[2^i]{n^d}\right) \quad .$$

If, on the other hand, the maximum recursion depth possible, is chosen, we get a query complexity of $O(\log^d n)$ with respect to n and an update complexity of $O(\log^d n)$ with respect to n, which is the same as the complexity of the SDDC [RAAP00]. Therefore,

Corollary 1. *If the trade-off parameter, $i = O(\log\log n)$, both the update and range-sum complexity becomes $O(\log^d n)$ (This matches the complexity of the Space-Efficient Dynamic Data Cube).*

Corollary 2. *If the trade-off parameter $i = l$, for some constant l, the range-sum complexity is $O(2^{ld})$, which is constant with respect to n, and the update complexity is*

$$O(\sqrt[2^i]{n^d}) = o(n^d) \quad .$$

2.2.3 Comparison to Previous Structures

Our structure has better asymptotic query-update trade-off compared to previous structures. To our best knowledge, the best previously known constant- query-time structure that does not require extra memory is the SRPS that has an update complexity of $O(\sqrt{n^d})$. For constant query complexity, our structure has $O(\sqrt[2^i]{n^d})$ update complexity. The SDDC, with our proposed trade-off has a query complexity of $O(n^d)$, for constant query complexity. The SDDC was designed with query and update complexities of $O(\log^d n)$. This can be, asymptotically, matched by our structure by choosing the trade-off parameter, $i = \log \log n$.

Additionally, our structure can be easier to parallelize, because the data cube is divided by $O(\sqrt{n})$ instead of blocks of constant size.

2.3 Analysis of Space-Efficient Dynamic Data Cube

In this section, we will present an analysis of the SDDC algorithm [RAAP00]. Furthermore, we will introduce a trade-off parameter for the SDDC, which is not introduced in the original SDDC. It is important to observe that the trade-off parameter is the recursion depth in both our structure and the SDDC. Our structure, however, requires a lower recursion depth to achieve the same performance.

Theorem 3. *The range-sum complexity of the SDDC is $\Omega(i^d)$, where i is the recursion depth and d is the dimension of the data cube.*

Proof. Let $T_d(n, i)$ denote the range-sum complexity of SDDC. Since the SDDC algorithm reduce the problem size by a factor of k after each recursion step, we have

$$T_d(n, i) = \sum_{j=0}^{d} \binom{d}{j} T_j(n/k, i-1), \text{if } i > 0 \text{ and } d > 0$$

and

$$T_d(n, i) = 1, \text{if } d = 0 \text{ or if } i = 0 \ .$$

We will prove that

$$T_d(n, i) \geq i^d$$

by induction on d.
For $d = 1$, we know that

$$T_1(n, i) = i + 1 \geq i \ .$$

Assume that

$$\forall d' < d : T_{d'}(n, i) \geq i^{d'} \ .$$

Hence,

$$\begin{aligned}
T_d(n,i) &= \sum_{j=0}^{d} \binom{d}{j} T_j(n/k, i-1) \\
&\geq \sum_{j=0}^{d-1} \binom{d}{j} (i-1)^j + T_d(n/k, i-1) \\
&= (i-1+1)^d - (i-1)^d + T_d(n/k, i-1) \\
&= i^d + (i-i)^d + T_d(n/k^i, i-i) \\
&= i^d + 1 \\
&\geq i^d \ .
\end{aligned}$$

□

The update time of the SDDC algorithm is as follows.

Theorem 4. *The update complexity of the SDDC is $\Omega(n^d/k^{di} + k^d i^d)$, where i is the recursion depth and d is the dimension of the data cube.*

Proof. Let $T_d(n,i)$ denote the update complexity of SDDC. Since the SDDC algorithm reduce the problem size by a factor of k after each recursion step, we have

$$T_d(n,i) = \sum_{j=0}^{d} \binom{d}{j} T_j(n/k, i-1) k^{d-j}, \text{if } i > 0 \text{ and } d > 0$$

and

$$T_d(n,i) = n^d, \text{if } i = 0 \ .$$

We will prove that

$$T_d(n,i) \geq (n/k^i)^d + (ki)^d$$

by induction on d.
For $d = 1$, we have

$$\begin{aligned}
T_1(n,i) &= k + T_1(n/k, i-1) \\
&= 2k + T_1(n/k^2, i-2) \\
&= ik + \frac{n}{k^i} \ .
\end{aligned}$$

Assume that

$$\forall d' < d : T_{d'}(n,i) \geq (n/k^i)^{d'} + (ki)^{d'} \ .$$

Hence,

$$\begin{aligned} T_d(n,i) &= \sum_{j=0}^{d} \binom{d}{j} T_j(n/k, i-1)k^{d-j} \\ &= \sum_{j=0}^{d-1} \binom{d}{j} T_j(n/k, i-1)k^{d-j} + T_d(n/k, i-1) \ . \end{aligned}$$

Consider the first term in this equation. We have

$$\begin{aligned} &\sum_{j=0}^{d-1} \binom{d}{j} T_j(n/k, i-1)k^{d-j} \\ &\geq \sum_{j=0}^{d-1} \binom{d}{j} \left(\left(\frac{n}{k^{i-1}} \right)^j + k^j (i-1)^j \right) k^{d-j} \\ &= k^d \sum_{j=0}^{d-1} \binom{d}{j} \left(\left(\frac{n}{k^i} \right)^j (i-1)^j \right) \\ &= k^d \left(1 + \frac{n}{k^i}\right)^d - \left(\frac{n}{k^{i-1}}\right)^d + k^d(1+i-1)^d - k^d(i-1)^d \\ &= \left(k + \frac{n}{k^{i-1}}\right)^d - \left(\frac{n}{k^{i-1}}\right)^d + k^d i^d - k^d(i-1)^d \\ &\geq k^d i^d - k^d(i-1)^d \ . \end{aligned}$$

By telescoping, we obtain

$$\begin{aligned} T_d(n,i) &\geq k^d(i^d - (i-1)^d) + T_d\left(\frac{n}{k^i}, i-i\right) \\ &= k^d i^d + \left(\frac{n}{k^i}\right)^d \ . \end{aligned}$$

□

2.4 Further Improvements

The memory accesses of our structure (and other similar structures) are quite scattered, which could lead to poor performance on cache systems or on secondary storage.

We can overcome most of this problem by reorganizing the memory layout of the algorithm. In our structure, we suggest storing all the border cells from blocks associated with one recursion call in the same place. This is automatically performed for the inner cells (since they already are clustered into the inner cells). Instead of storing the border cells in their respective block, we suggest storing the border cells first (at low index) and then storing all the inner cells in blocks that are one element

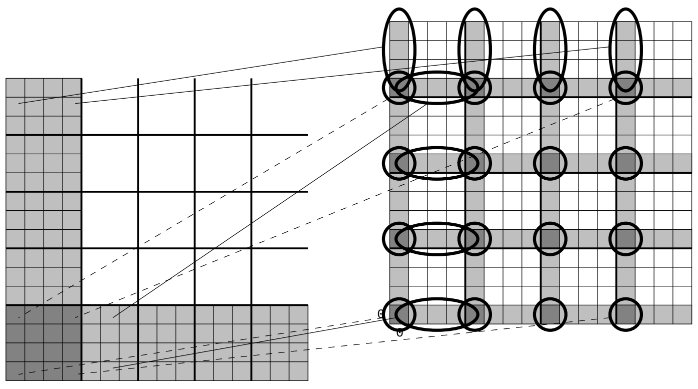

Figure 2.7: Efficient space localization

smaller than before (because of the missing border cells). This should be performed for each recursion level. This way, the data access becomes more concentrated to the same region. See Figure 2.7 for a two dimensional example.

Chapter 3

Maximum Sum Subsequences

In this chapter, we deal with the problem of finding the k maximum sum subsequences. The problem investigated generalizes the maximum subsequence problem, where $k = 1$. The latter problem has been well studied because of its applications in many areas and is an often-cited example in algorithm design.

The problem can be defined as follows: Given a sequence X of real numbers $X = \langle x_1, x_2, \cdots, x_n \rangle$ and an integer k, $1 \leq k \leq \frac{1}{2}n(n-1)$, the *$k$ maximum sum subsequences problem* is to select k pairs of indices

$$\{(i_\ell, j_\ell) : 1 \leq i_\ell \leq j_\ell, \ell = 1, 2, \cdots, k\}$$

such that the (range) sums

$$\sum_{p=i_\ell}^{j_\ell} x_p, \ell = 1, 2, \cdots, k,$$

are the k largest values among all the possible sums

$$\sum_{\ell=i}^{j} x_\ell, 1 \leq i \leq j \leq n \ .$$

Notice that we do not require that the sums are sorted. The algorithms presented in this chapter compute only the range sums without pointing out the subsequences explicitly; extending the algorithms to output the subsequences computed as well is straightforward.

The k maximum sum subsequences problem becomes trivial when $k = \Theta(n^2)$: In this case, an optimal algorithm for the problem runs in $\Theta(n^2)$ time in the worst case. To achieve this, first compute all possible sums

$$\sum_{\ell=i}^{j} x_\ell, (1 \leq i \leq j \leq n)$$

and then select the k^{th} largest sum. Then, traverse the list of all the possible sums and pick all sums larger than or equal to the k^{th} sum in order to get all the k largest sums. For arbitrary value of the parameter k, we first propose an algorithm running in $O(k + n\log^2 n)$ time in the worst case. Then, we show that the problem can also be solved in $O(n\sqrt{k})$ time; which is even faster for small values of the user-specified parameter k. Combining these two algorithms results in a worst-case running time

$$O\left(\min\left\{k + n\log^2 n, n\sqrt{k}\right\}\right) \quad .$$

This time complexity is the best possible for $k \geq n\log^2 n$. Previously, the optimal methods were known only for the extreme case: $k = 1$. For arbitrary value of k, the previously best known result for this problem is $\Theta(nk)$ [BT04] for $1 \leq k \leq \frac{1}{2}n(n-1)$. Our algorithms are faster for any value of k.

The 2-dimensional version of the *k maximum sum subsequences problem* is as follows: Given two-dimensional array of order $m \times n$, find k orthogonal continuous subregions such that each has a sum at least as large as the k^{th} largest sum. The sums are chosen from the set of all continuous subarrays of the given array. A straightforward solution would cost $\Theta\left(m^2n^2\right)$ via an enumeration of all possible sums. We will show how to reduce the time complexity for this problem to $O(\min\{m^2C, m^2n^2\})$ in the worst case , where $C = \min\{k + n\log^2 n, n\sqrt{k}\}$. Extending our techniques to higher dimensions results in a solution requiring $O(n^{2d-2}\min\{C, n^2\})$ time for a given d-dimension array with size n in each dimension.

In processing our algorithms for the above problems, we need to perform the selection and the search operation in the *Cartesian Sum* $A + B$ of a sequence $A = \langle a_1, a_2, \cdots, a_m\rangle$ and a sequence $B = \langle b_1, b_2, \cdots, b_n\rangle$ of real numbers, where

$$A + B = \{a_i + b_j | 1 \leq i \leq m \text{ and } 1 \leq j \leq n\}$$

Optimal algorithms have been developed for those problems [FJ82, FJ84]. However, those algorithms are not applicable here, because of special conditions on the solution imposed by our problems. What we are interested in are some elements with the Cartesian sum set with particular indices. Define the set, $A \bigtriangleup B$, of all the *good elements* in $A + B$ is defined as

$$A \bigtriangleup B = \{a_i + b_j | 1 \leq i \leq j \leq n\}$$

assuming that $n = m$. The elements of $A + B$ that are not good will be referred to as *bad elements.*

Definition 1. *The* rank *of an element x in a given set A is defined as*

$$\texttt{rank}(x; A) \triangleq \|\{a | a \in A, a \leq x\}\| \quad .$$

For a given matrix $M = [a_{i,j}]$ of order $n \times m$,

$$\texttt{rank}(x; M) \triangleq \|\{a_{i,j} | a_{i,j} \leq x, 1 \leq i \leq n, 1 \leq j \leq m\}\| \quad .$$

Definition 2. *The* relative rank*, $\texttt{rank}_G(x; M)$, of x with respect to the* good elements *in a matrix $M = [a_{i,j}]_{n \times n}$ is defined as*

$$\texttt{rank}_G(x; M) \triangleq \|\{a_{i,j} | a_{i,j} \leq x, 1 \leq i \leq j \leq n\}\| \quad .$$

Throughout the presentation we assume that either the input array is already resident in internal memory, or each element can be computed as needed in constant time. All logarithms are to the base 2.

3.1 The General Algorithm

In this section, we present an algorithm for solving the k maximum sum subsequences problem on a given sequence of length n. Our algorithm consists of five phases.

1. In the first phase, the problem is transformed to the problem of finding the top k maximum values over all the good elements in a particular matrix of order $n \times n$.

2. The second phase performs repeated constraint searches which decreases the number of candidate elements to $O(\min\{kn, n^2\})$.

3. A procedure of range reduction will be carried out in the third phase. This procedure is able to reduce the number of candidates further to $\Theta(k)$.

4. In the fourth phase, a worst-case linear-time selection algorithm is run on the remaining candidates. The output of this phase is an element, x, that is the k^{th} largest range sum.

5. The final phase involves finding the good elements whose values are not less than x.

In the following sections, we will show how to peruse these phases in detail.

3.1.1 Problem Transformation

Given an input instance $X = \langle x_1, \cdots, x_n \rangle$ of the k maximum sum subsequences problem, we can first construct the prefix-sum sequence $P = \langle p_1, \cdots, p_n \rangle$ of X:

$$\begin{aligned} p_1 &= x_1 \\ p_{i+1} &= p_i + x_{i+1} \text{ for } 1 \le i \le n-1 \ . \end{aligned}$$

This can be done in $O(n)$ time in the worst case. It is then clear that any range sum $\sum_{l=i}^{j} x_l$ of the sequence X is equal to $p_j - p_{i-1}$:

$$\sum_{l=i}^{j} x_l = p_j - p_{i-1} \ ,$$

for $i \le j$. Define a new array $Q = \langle q_1, q_2, \ldots, q_n \rangle$ where $q_i = -p_{i-1}$ for $1 \le i \le n$. Hence, the goal of the k maximum sum subsequences problem becomes finding the k largest values among all the good elements in the Cartesian sum set $Q + P$. Although selection, search, and ranking problems in matrices and Cartesian sum sets are well studied, previous algorithms on matrices and Cartesian sum sets are not directly applicable here. This is because of the fact that we are only interested in the good elements. Notice that every element of $Q + P$ does not represent a valid subsequence sum of the original input sequence. Only the good elements do. The bad elements do not represent positive-length subsequences, but negative-length subsequences and may, therefore, not be included in the final result. In the subsections that follow, we will solve the selection, search, and ranking problems in Cartesian sum sets, which are of their own interests. Given two sequences A and B, we assume for simplicity that all the elements in $A + B$ are distinct and our algorithms operate on the matrix M of the form $A + B$. Notice that M is not actually constructed. We will describe our algorithms for the k smallest elements in the sets under consideration. It is straightforward to adapt the algorithms when the largest elements are requested.

3.1.2 Constraint Search and Ranking

After the problem transformation, the number of candidate elements is $\Theta(n^2)$. In this section, we will show how to reduce the number of candidates to $O(\min\{kn, n^2\})$ by performing constraint search and ranking.

Let $A = \langle a_1, a_2, \cdots, a_n \rangle$ and $B = \langle b_1, b_2, \cdots, b_n \rangle$ be two sequences of numbers. Consider the problem of computing the number of bad elements in $A + B$ that are less than or equal to a given number x. Namely, we are interested in calculating the value $\mathtt{rank}(x; A + B) - \mathtt{rank}_G(x; A + B)$. This rank computation can be done by a search in $A + B$ with respect to the good elements. In order to do so, let $\hat{A}$ be the sorted sequence (in non-decreasing order) of A.

Notice that each column in the matrix $M = \hat{A} + B$ is sorted. What we want to do is to count the number of good elements less than or equal to x in each column of M. We first perform a binary search on each column j to compute the number of elements less than or equal to x in that column. Let i_j be the largest index such that $M[i_j, j] \leq x$. That is, all elements above i_j in column j are smaller than or equal to x. Since we have not rearranged or reordered the sequence B, the n^{th} column of M contains no bad elements. Column $n-i$ will contain i bad elements for $1 \leq i \leq n-1$.

Observation 1. *Column j, $1 \leq j \leq n$, in the matrix M has $n-j$ bad elements.*

The relative positions of the bad elements among all the columns can be decided by the following observation:

Observation 2. *If column j in the matrix has a bad element at row i, then every column ℓ, $1 \leq \ell \leq j-1$, will have a bad element at row i as well.*

In fact, if $M[i,j] = \hat{A}[i] + B[j]$ is bad and $\hat{A}[i]$ is the element $A[i']$, then $i' > j$. Hence, any element $M[i,\ell] = \hat{A}[i] + B[l]$ is bad if $i' > j > \ell$ for $1 \leq \ell \leq j-1$. To locate the position of these bad elements with respect to positions $\{i_1, \cdots, i_j\}$, we scan the matrix column by column from the right to the left. Moreover, we need only to process one bad element at a new position in each column when going left.

For this purpose, let $\pi(i)$ be the sorted position of $A[i]$ for $i = 1, 2, \ldots, n$; i.e,

$$A[i] = \hat{A}[\pi(i)] \text{ for } i = 1, 2, \cdots, n \ .$$

We will construct an array, L, of length n to store the information about the positions and the number of bad elements when scanning columns of M from the right to the left. Initially, L is an array of zeros. Consider one column at a time. Assume that the column currently accessed is the j^{th} column. Upon processing columns $n, n-1, \cdots, j+1$, we will assign $L[\pi(j+1)] = 1$ since the current column j has a bad element at row $\pi(j+1)$ (that is, one different row from all the n^{th} to the $(j+1)^{\text{th}}$) column. Hence, L has value 1 at the positions corresponding to the rows where the elements are bad for the current column. Therefore, the number of bad elements in column j that is less than or equal to x is

$$n_j = \sum_{i=1}^{i_j} L[i] \ ,$$

where $L[i]$ is the current value stored of the array position i of L just after processing column j for $j = n-1, n-2, \ldots, 2, 1$. The only difference between column $j+1$ and column j is that column j contains one more bad element. As mentioned above, the location of this bad element can be computed with the help of the permutation π in constant time. Moreover, the computation of n_j can be done in $O(\log n)$ time using an algorithm for computing dynamic prefix sums, for example our algorithm from chapter 2 or the algorithm by Riedewald et al. [RAA01]. The number of bad elements computed for each column could be accumulated as an integer number over all columns. Since the number of columns is n and each column requires $O(\log n)$ time which gives a total time of $O(n \log n)$ for the whole matrix M. The pseudo-code for finding all the bad elements that are less than or equal to x is as follows:

```
1. Initiate the array $L$, to represent an array of zeros. (Initiate the data structure
   for the dynamic prefix sum to represent an array of zeros.)
2. Let $l = 0$.
3. For $j = n - 1$ downto 1 do
   (a) Locate the position for $x$ in column $j$ by a standard binary search.
   (b) Let $i_j$ be the largest index such that $M[i_j, j] \leq x$ .
4. For $j = n$ downto 1 do
   (a) Compute
           $$s = \sum_{i=1}^{i_j} L[i]$$
       using the algorithm for dynamic prefix sum.
   (b) Let $l = l + s$ .
   (c) Let $L[\pi(j)] = 1$ and update the data structure by employing the algorithm
       for dynamic prefix sum.
5. Return $l$.
```

The above algorithm operates on the matrix M. Since the matrix M requires that the sequence A is sorted and sorting A sorting requires $O(n \log n)$ time, we have proven the following:

Lemma 1. *Let $A = \langle a_1, a_2, \cdots, a_n \rangle$ and $B = \langle b_1, b_2, \cdots, b_n \rangle$ be two sequences of numbers. Given a number x, we can compute the number of bad elements in $A + B$ whose values are less than or equal to x in $O(n \log n)$ time in the worst case.*

By letting X in the above lemma be the k^{th} largest element in $A + B$, we can reduce the number of candidate element for the k maximum elements from $A + B$ to $\min\{kn, n^2\}$.

3.1.3 Range Reduction and Listing

In order to decrease the number of candidates further to $\Theta(k)$, we need first to study the problem of finding and listing out all the good elements in $A+B$ that are less than or equal to a given number x, where $A = \langle a_1, a_2, \cdots, a_n \rangle$ and $B = \langle b_1, b_2, \cdots, b_n \rangle$ are two given sequences of numbers. The fast solution to this problem is applied in our solution to the k maximum sum subsequences problem. This will result in that the range of the candidates is decreased. The procedure is repeated until the sub-problems can be solved in desired time bounds.

Similarly to the previous subsection, consider the matrix

$$M = \hat{A} + B$$

where $\hat{A}$ is the sorted output when sorting A in non-decreasing order. In this case, each column of M is sorted. As discussed above, each column, except the rightmost one, will contain a number of bad elements. Moreover, column j can contain one and only one more bad element than column $j+1$. Again, this bad element can be found with the help of the sort permutation, π, of A. That is,

$$A[i] = \hat{A}[\pi(i)] \text{ for } i = 1, 2, \cdots, n \ .$$

For a fast enumeration of the desired good elements, we will use two extra lists: A list L for storing all the good elements searched for and a doubly linked list of length n, called *jump list*, in order to jump over all the bad elements during the search. It is crucial that the bad elements are not scanned in order to achieve the desired time complexity, since there can be $\Theta(n^2)$ of them. We will implement the jump list with two arrays of indices, the *forward array*, F, and the *backward array*, G. Each element of the forward array is the index of the next good element in the current column. Each element in the backward array is the index of the previous good element in the same column. The same forward and backward arrays will be used for each column, but updated appropriately. Note that the number of bad elements only differ by one between adjacent columns.

Starting from the rightmost column, we will process the columns of the matrix as follows: Begin with the smallest element of the column, if it is less than or equal to x, add it to L and continue to process the next good element in the same column. If not, we are done with the column.

Initially, $F[i] = i+1$ and $G[i] = i-1$. Before listing and storing the good elements ($\leq x$) in the current column, we compute the new bad element for this column (by using π). Notice that the number of such bad elements is at most one. Then we update F and G as follows: Let the index of the new bad element in the current column be i. Now, we want to remove the bad element from the linked list represented by the two arrays , F and G. Therefore, we change the forward pointer of the element previous to element i, which has index $G[i]$, in the forward array, so that it points to the element following element i in the forward array. The backward array is changed similarly.

Let $F[G[i]] = F[i]$ and $G[F[i]] = G[i]$. We search for good elements smaller than or equal to x in the order of the positions $F[1], F[F[1]], F[F[F[1]]], \ldots$ in the current column. Assume that the number of good elements in column j whose values are less than or equal to x is r_j. Thus, the time used in processing column j will be $O(r_j)$ plus a constant amount of work (for F, G, and an extra element scanned), which is $O(r_j + 1)$. The total time required will, therefore, equal

$$O\left(\sum_{j=1}^{n}(r_j + 1)\right) = O\left(\mathtt{rank}_G(x; A + B) + n\right) \ .$$

To sum up, the above algorithm for enumerating the desired good elements can be presented by the following pseudo-code:

```
1. Initiate $F[i] = i + 1$, $0 \le i \le n - 1$.
2. Initiate $G[i] = i - 1$, $1 \le i \le n$.
3. Initiate $L$ = empty list.
4. For $j = n$ downto 1 do
   (a) Let $i = F[0]$.
   (b) While $M[i, j] \le x$ do
       i. Let $L \leftarrow L \cup M[i, j]$ .
       ii. Let $i = F[i]$ .
   (c) Let $i = \pi(j + 1)$ .
   (d) Let $F[G[i]] = F[i]$ and $G[F[i]] = G[i]$ .
```

The above algorithm requires the sorting of A. Since sorting the sequence A takes $O(n \log n)$ time, we have

Lemma 2. *Let A and B be two sequences of numbers each of length n. Given a number x, finding all the good elements in $A + B$ that are less than or equal to x takes time at most*

$$O(\mathtt{rank}_G(x; A + B) + n \log n) \ .$$

3.1.4 Putting Things Together

Now, we are ready to present our algorithm for the k maximum sum subsequences problem. Given a sequence $X = \langle x_1, \cdots, x_n \rangle$ of numbers and an integer $k \ge 1$.

1. If $k = \Theta(n^2)$, then

   (a) Enumerate all the possible range sums of X.

```
    (b) Select the $\min\{k, \frac{1}{2}n(n-1)\}$ largest ones.
    (c) Exit.
2. Compute the prefix sums sequence, $P$, of $X$:
```

$$P = \langle p_1, \cdots, p_n \rangle \quad ,$$

```
   where
```

$$\begin{aligned} p_1 &= x_1 \\ p_{i+1} &= p_i + x_{i+1}, \text{ for } i = 1, \cdots, n-1 \quad . \end{aligned}$$

```
3. Let $Q = \langle q_1, q_2, \ldots, q_n \rangle$ , where $q_i = -p_{i-1}$ for $i = 1, 2, \ldots, n$ and $p_0 \triangleq 0$.
4. (a) Sort $Q$ in non-decreasing order; denote the sorted output $A$.
   (b) Sort $P$ in non-decreasing order; denote the sorted output $B$.
5. (a) Let $a \leftarrow 1$.
   (b) Let $b \leftarrow k$.
6. Repeat
   (a) Select the $b^{th}$ largest element, $x$, in A+B.
   (b) Compute the number, $m$, of the good elements in $Q + P$ that is $\geq x$.
   (c) If $m < k$, then
        i. let $a \leftarrow b$, and
       ii. let $b \leftarrow 2b$.
   until $m \geq k$.
7. while $m > 2k$ do
   (a) $x \leftarrow \left[ \text{the } \lfloor \frac{a+b}{2} \rfloor^{th} \text{ largest element in } A + B \right]$.
   (b) $m \leftarrow [$ the number of good elements $\geq x$ in $Q + P]$.
   (c) if $m < k$ then
            $a \leftarrow \left\lfloor \frac{a+b}{2} \right\rfloor$
       else
            $b \leftarrow \left\lfloor \frac{a+b}{2} \right\rfloor$
```

8. Let S be the set of all good elements in $Q+P$ whose values are greater than or equal to x.
9. Find all the k largest elements in S.

After the problem transformation, our algorithm searches for an element, x, by repeated range reductions so that the relative rank of x in $Q+P$ is between k and $2k$. Next, the algorithm discards all the good elements whose values $\geq x$ and all the bad elements in $Q+P$. Then, the k maximum range sums of X are the first k largest values among the remaining elements.

3.1.5 Complexity Analysis

The above algorithm runs in $O\left(k + n\log^2 n\right)$ time in the worst case. In fact, Step 1 is done in $O(n^2) = O(k)$ time. Steps 2 and 3 run in linear time. Step 4 takes $O(n\log n)$ time.

During each iteration of Step 6, (a) can be performed with $O(\sqrt{b}) = O(n)$ operations [FJ84] since both A and B are sorted; (b) takes $O(n\log n)$ time with the help of the sorted sequence A of Q by Lemma 1. The number of iterations can easily be computed from the following lemma.

Lemma 3. *Given an element x and a set, A, of N elements: If* $\texttt{rank}(x;A) = r$*, then* $r \leq \texttt{rank}(x;B) \leq N+r$*, where* $B = \{b | b \in A \text{ or } -b \in A\}$.

Proof. Let all the elements in A whose values are less than or equal to x be $a_1, a_2, \cdots$ $\cdots, a_r$. Assume, without loss of generality, that all of $a_1, a_2, \cdots, a_i$ are strictly less than 0 and all of $a_{i+1}, a_{i+2}, \cdots, a_r$ are greater than or equal to 0, where $1 \leq i \leq r$. Obviously,

$$\texttt{rank}(x;B) \geq r$$

for the set B defined in the lemma. On the other hand, the following set:

$$\{b | b \in B \text{ and } b \leq x\} \ ,$$

contains at most

- $a_1, a_2, \ldots, a_r$
- $-a_{i+1}, -a_{i+2}, \ldots, -a_r$
- $-a_1, -a_2, \ldots, -a_i$
- $-a_{r+1}, -a_{r+2}, \ldots, -a_n$

Therefore, the size of $\{b | b \in B \text{ and } b \leq x\}$, is at most $r+(r-i)+i+(N-r) = N+r$. The lemma follows. □

Notice that in Step 6(c), the value of b is doubled each time. Therefore, when Step 6 is done, we have $a = 2^\ell k$ and $b = 2^{\ell+1}k$ for some integer $\ell \geq 0$. Let

$$N = \frac{1}{2}n(n-1)$$

and $r = k$. From the above lemma, we have

$$2^\ell k \leq N + k$$

and thus,

$$\ell = O\left(\log \frac{N}{k}\right) = O(\log n) \ .$$

Hence, Step 6 takes at most $O(n \log^2 n)$ time.

Step 7 performs a binary-search-like operation, where each step does a selection and a computation of relative rank. Similar to that of Step 6, the time complexity of this step equals $O(n \log n \log(b-a)) = O(n \log^2 n)$. Step 8 requires $O(\texttt{rank}_G(x; Q + P)) = O(k)$ from Lemma 2 since we have already sorted the sequence Q before this step. Finally, Step 9 takes $O(k)$ time as well.

Theorem 5. *The k maximum sum subsequences problem can be solved in $O(k + n\log^2 n)$ time in the worst case.*

3.2 Fast Computation When k is Small

An obvious lower bound for the k maximum sum subsequences problem is $\Omega(k + n)$. Thus, the algorithm presented in the previous section gives the best possible running time when $k = \Omega\left(n \log^2 n\right)$. However, the algorithm is not the fastest one for small values of k. Observing that if the input sequence is divided into two blocks of consecutive elements, all the subsequences having the k largest range sums must be entirely located in the left block, in the right block, or cross the border between the left and the right block. By crossing the border we mean that the left endpoint of the subsequence is located in the left block and the right endpoint in the right block.

In order to efficiently find the range sums crossing the border between the blocks, we will precompute all the suffix sums S of the left block and all the prefix sums P of the right block. After that, all the range sums crossing the border are now elements in the Cartesian sum of $S+P$. The k largest elements in $S+P$ can be found fast if we first sort the sets S and P, respectively. In fact, our algorithm produces not only the k maximum sum subsequences, but also the k largest prefix sums, the k largest suffix sums (in sorted order), and the sum of the given sequence. That is, our algorithm indeed solves the following problem:

maxSum(X,k):	
Input:	A sequence X of n reals and an integer k, $1 \leq k \leq \frac{1}{2}n(n-1)$.
Output:	K_X: the set of the k largest range sums of X
	P_X: the set of the $\min\{k,n\}$ largest prefix sums of X
	S_X: the set of the $\min\{k,n\}$ largest suffix sums of X
	$w(X)$: the sum of all the elements in X

Notice first that if $k = \Theta(n^2)$, the above problem can be solved by first computing all the possible range sums of X and then select the k largest ones. This can be done in $\Theta(n^2)$ time in the worst case. Similarly, $w(X)$, P_X and S_X can be computed in $O(n \log n) = O(n^2)$ time. Therefore, we have

Lemma 4. *The k maximum sum subsequences problem, when $k = \Theta(n^2)$, can be solved in $\Theta(n^2)$ time in the worst case.*

For an arbitrary value of k, the above observations lead us to a divide-and-conquer approach to solving the given problem. However, in order to speed up the computation, the recursion will be forced to stop when the size of the underlying block drops below $\Theta(\sqrt{k})$. This is because when $k = \Theta(n^2)$, we will have to calculate almost every possible range-sum and this is done faster using the trivial algorithm. In that case, we will apply the method from the above lemma to that block.

```
Algorithm: AlgoMaxSum($X,k$)
Input:    A sequence $X$ of $n$ real numbers and an integer $k \geq 1$
Output:   $(K_X, P_X, S_X, w(X))$

1. If $k = \Theta(n^2)$ then
   (a) let
         $k' = \min\left\{k, \frac{1}{2}n(n-1)\right\}$ ,
   (b) compute $(K, P, S, w)$ according to Lemma 4 with the parameter integer $k'$.
   (c) Exit.
2. Divide $X$ into two blocks of equal size:
         $L = X[1..n/2]$
   and
         $R = X[n/2+1..n]$ .
```

3. Recursively solve the problem *maxSum* on L and R, respectively, and let
   (a) $(K_L, P_L, S_L, w(L)) \leftarrow \texttt{AlgoMaxSum}(L, k)$,
   (b) $(K_R, P_R, S_R, w(R)) \leftarrow \texttt{AlgoMaxSum}(R, k)$.
4. Find the set, C, of the k largest elements in $S_L + P_R$.
5. Select the k largest elements in $K_L \cup C \cup K_R$ resulting in the set K_X.
6. Assign the k largest elements in $P_L \cup (P_R + \{w_L\})$ to P_X.
7. Assign the k largest elements in $(S_L + \{w_R\}) \cup S_R$ to S_X.
8. Compute $w(X) = w(L) + w(R)$.

The correctness of the algorithm follows directly from the above description and the observation at the beginning of this section. We will analyze the time complexity step by step. Let $T(n, k)$ be the worst-case running time of the algorithm $\texttt{AlgoMaxSum}(X, k)$. In the case when $k = \Omega(n^2)$, the algorithm (now Step 1) costs $O(n^2)$ time. Step 2 takes $O(1)$ time and Step 3 requires $2T(n/2, k)$ time. Step 4 can be done by first finding the k^{th} largest element, x, in $S_L + P_R$ and then searching for all the elements in $S_L + P_R$ that are larger than x. Here we may assume without loss of generality that all the elements in $S_L + P_R$ are distinct. Notice that both S_L and P_R are sorted. Thus, x can be found in $O(k)$ time [FJ82] and the search procedure above can be solved in $\Theta(k)$ time [FJ84]. Step 5 takes $\Theta(k)$ time using any standard worst-case linear time selection algorithm. One way to implement Step 6 is to first merge P_L and $P_R + \{w_L\}$ and then choose the k largest elements in the resulting sorted set, which can be done in $\Theta(k)$ time. Similarly, Step 7 runs in $\Theta(k)$ time. The last step needs only constant time. To sum up, we have the following recurrence

$$T(n, k) = \begin{cases} 2T\left(\frac{n}{2}, k\right) + O(k) & , \text{if } k = o(n^2) \\ O(n^2) & , \text{if } k = \Omega(n^2) \end{cases} .$$

With the substitution method, we have

$$T(n, k) = 2^i T\left(\frac{n}{2^i}, k\right) + O\left(\sum_{j=0}^{i-1} k 2^j\right) .$$

Letting $(n/2^i)^2 = k$ results in $2^i = n/\sqrt{k}$. Hence,

$$\begin{aligned} T(n, k) &= \left(n/\sqrt{k}\right) T\left(\sqrt{k}, k\right) + O(n\sqrt{k}) \\ &= \left(n/\sqrt{k}\right) O(k) + O(n\sqrt{k}) = O(n\sqrt{k}) . \end{aligned}$$

Therefore,

Theorem 6. *The k maximum sum subsequences problem can be solved in $O(n\sqrt{k})$ time in the worst case.*

Combining Theorems 5 and 6, we have

Corollary 3. *Given a sequence of length n and an integer parameter, k, the k maximum sum subsequences problem can be solved in $O(\min\{k + n\log^2 n, n\sqrt{k}\})$ time in the worst case.*

In the next section, we will show how to solve the high-dimensional problems incrementally by using our fast algorithms developed in Sections 3 and 4.

3.3 Higher Dimensional Cases

We first study the two-dimensional version of the problem; that is, to find all the orthogonal continuous subregions whose sums are at least as large as the k^{th} largest sum among all continuous subarrays of a given two-dimensional array. The following algorithm simply reduces the dimension parameter in the problem. Actually, for a given array

$$X = [x_{i,j}]_{1\le i\le m, 1\le j\le n}$$

of real numbers, we transform the problem into $\Theta(m^2)$ one-dimensional k maximum sum subsequences problems, and the later problems are solved using our one-dimensional algorithms.

```
Algorithm: AlgoMaxSum2D($X, k$)
Input:     A two-dimensional array

               $X = [x_{i,j}]_{1\le i\le m, 1\le j\le n}$

           of real numbers and an integer $k \ge 1$ .
Output:    The set $K_X$ of the $k$ largest sums among all possible sums

               $\sum_{i_1=j_1}^{j_2} \sum_{i_2=j_3}^{j_4} x_{i_1,i_2}$

           for $1 \le j_1 \le j_2 \le m, 1 \le j_3 \le j_4 \le n$ .
  1. Compute a new array,

         $Y = [y_{i,j}]$

     of order $m \times n$, where

         $y_{i,j} = \sum_{l=1}^{j} x_{i,l}$ .
```

2. For each i and j, $1 \leq i \leq j \leq m$,
   (a) Create an array
   $$A_{i,j} = \langle a_1, \cdots, a_n \rangle$$
   such that $a_l = y_{j,l} - y_{i-1,l}$ for $l = 1, 2, \cdots, n$.
   (b) Solve the k maximum sum subsequences problem on $A_{i,j}$ with the parameter k; output $K_{A_{i,j}}$.
3. Let K_X be the k largest elements in the union of $K_{A_{i,j}}$ for $1 \leq i \leq j \leq m$.

Clearly, this algorithm computes the k largest range sums. Notice that Step 1 actually computes the prefix sums for each column of X; which can be done in $\Theta(mn)$ time. Step 2 requires $O(C)$ operations for every fixed i and j from Corollary 3, where $C = \min\{k + n\log^2 n, n\sqrt{k}\}$. Since the total number of index combinations is $\Theta(m^2)$, the time complexity of this step is thus $\Theta(m^2C)$. Note that the size of the union of $K_{A_{i,j}}$, $1 \leq i \leq j \leq m$, is $\Theta(m^2k)$. Hence, the selection (Step 3) requires $\Theta(m^2k)$ time in such a set. Therefore, the algorithm `AlgoMaxSum2D` takes $O(m^2C + m^2k)$ time. If $k = o(n^2)$, then $m^2C + m^2k = O(m^2n\sqrt{k})$. If $k = \Omega(n^2)$, however, we can run a straightforward algorithm running in $O(m^2n^2)$ time that enumerates all possible range sums and selects the k largest ones, instead of `AlgoMaxSum2D`. Therefore, we have

Theorem 7. *Given a two-dimensional array, X, of order $m\times n$ and an integer, $k \geq 1$, the k maximum sum subarrays problem can be computed in $O(\min\{m^2C, m^2n^2\})$ in the worst case, where $C = \min\{k + n\log^2 n, n\sqrt{k}\}$.*

An algorithm for the k maximum sum subarrays problem with a running time of $\Theta(m^2nk)$ has been presented in [BT04]. Notice that the value of the parameter k is between 1 and $\Theta(m^2n^2)$. Our algorithm improves over the above result for every value of k.

It is possible to extend our approach to the d-dimensional version of the maximum sum subarrays problem. Given a d-dimensional array of real numbers, with size n in each dimension, we can enumerate all the possible $\binom{n}{2}^{d-1}$ problems of the $(d-1)$-dimensional version. This will give a total cost of

$$O\left(\binom{n}{2}^{d-1} n\sqrt{k} + \binom{n}{2}^{d-1} k\right) =$$

$$O\left(n^{2d-1}\sqrt{k} + n^{2d-2}k\right)$$

from Theorem 6. Of course, if $k = \Omega(n^2)$, we may just run a straightforward algorithm, similar to that for the two-dimensional case. Thus,

Theorem 8. *The k maximum sum subarrays of a d-dimensional array of order $n_1 \times n_2 \times \cdots \times n_d$ can be found in*

$$O\left((n_1 \times n_2 \times \cdots \times n_{d-1})^2 n_d \min\left\{\sqrt{k}, n_d\right\}\right)$$

time in the worst case.

3.4 Conclusions

We have addressed the problem of computing k maximum sum subsequences/subarrays and proposed efficient algorithms. The bounds that we obtain improve substantially on previous results and our algorithms are optimal for some non-constant values of the parameter k ($k \geq n \log^2 n$). Previously, only for the extreme case when $k = 1$, the problem investigated could be solved optimally. We have made some progress in designing optimal algorithms. However, any algorithm that computes the k largest range sums requires at least $\Omega(n + k)$ operations in the worst case. It is an interesting problem to see whether this lower bound is achievable. Moreover, some new ideas will be needed in order to handle the higher-dimensional problems optimally.

Chapter 4

Range Top-k Queries

In this chapter, we design a hierarchical structure on data cubes that enables us to solve the two-dimensional range top-k query problem in linear time in the size of the query region, in the worst case. This improves over the *Adaptive Pre-computed Partition Top* (APPT) method by Loh et al. [LLAL02] significantly. For higher dimensions, our structure performs at least as good as that of the APPT. Moreover, a theoretical performance analysis of the APPT algorithm is also provided, which is omitted in the original paper.

Before describing the APPT algorithm and providing a worst-case complexity analysis, we first define the problem of range top-k query.

4.1 Range Top-k Query

In the same way as in previous chapters, we model a data cube (DC) with $d+1$ attributes as a d-dimensional array, A, of size $n_1 \times n_2 \times \cdots \times n_d$. Assume without loss of generality that $n_i = n$ for all $1 \leq i \leq d$. Given an integer $k \geq 1$ and a range specified by $\langle (a_1, b_1), \cdots, (a_d, b_d) \rangle$, the range top-$k$ query is to find the k largest elements among all the values stored in the cells $A[i_1, \cdots, i_d]$ for all $a_1 \leq i_1 \leq b_1$, $a_2 \leq i_2 \leq b_2$, $\cdots$, $a_d \leq i_d \leq b_d$. We will denote such a range top-k query by $Q(k; a_1, b_1; \cdots; a_d, b_d)$. For $k = 1$, the problem becomes the range-max problem which has been investigated and is well understood [KLMHL98, LLL00a, LLL00b, HAMS97]. Notice that k is a user-specified parameter.

If no data structure other than the data cube itself is used, solving the range top-k query problem becomes simple: Just visit all the elements in the query range and maintain a list containing all the k largest values, encountered so far. Since all the elements within the query range must be visited in the data cube, in order to correctly answer the query, the query cost will be $\Theta(m_1 \times m_2 \times \cdots \times m_d)$, where m_i is the size of the query interval along the dimension i (that is, $m_i = b_i - a_i + 1$) for $i = 1, 2, \cdots, d$. Notice that the query cost becomes $\Theta(n^d)$ when all the $m_i = \Theta(n)$. The space utilization for this straightforward method is, of course, the same as that for the data cube itself, plus the extra space for storing the answer, which is $\Theta(n^d + k)$. An update operation here takes only constant time, since there is no need to update any data structure; only the element of the data cube needs to be updated.

In order to reduce the query cost, at least in the average case, Loh et al [LLAL02] proposed a partition-based method, *Adaptive Pre-computed Partition Top* (APPT). The APPT method works as follows: In a preprocessing stage, the method first divides the data cube into blocks of equal size (the size is a constant number). Then the r largest elements within each block together with their indices are selected and stored in a data structure called the *Location Pre-Computed Cube* (LPC), for some integer $r > 0$. Moreover, the r^{th} largest element from each block is kept in a separate list as well. A cache, called *overflow array* is also allocated, where other elements of the block can be stored, resulting from the queries. Namely, if some query performed searched for the k' largest elements in this block and $k' > r$, then all the elements of rank between $r + 1$ and k' will be considered for storing in the cache. The purpose of this cache is to speed-up further queries, at least in the average case. The overflow array is common to all blocks and an area for a specific block is allocated from the overflow array as needed.

On handling a range top-k query, the APPT algorithm uses a sorted list, called *the candidate list* of size k, for storing the candidate answers to the query. Another sorted list, called *the guarding list* is used to determine from which blocks more elements than the r are needed and in what order to process the blocks. Note that the r^{th} largest value is the smallest value computed and stored for the block. Both lists are maintained sorted during the execution of the algorithm

First, the algorithm visits all the blocks covered (fully or partially) by the query range. On visiting a partition block, the algorithm inserts the r largest elements (if $k \geq r$; otherwise, the k largest elements are inserted) of the processed block from the LPC into the candidate list and the r^{th} largest element into the guarding list. If the number of blocks times r is larger than r, the smallest elements in the candidate list is thrown away as the list becomes full. For blocks only partially covered by the query region, the pre-computed elements located outside the query region is not inserted into the candidate list.

Then, the largest element, y, from the guarding list is compared to the smallest element, x, of the candidate list. If $y > x$ (the lists "overlap", in some sense), or if the candidate list does not contain k elements, then some more elements from the block corresponding y could also be candidates for the final answer. In this case, the overflow array is scanned and, if necessary, the data cube is scanned for elements. All the elements from the processed block (that is, y's block) larger than y will be added to the candidate list (if needed). The intention here is to keep an invariant: The size of the candidate list is k and the smallest value in the candidate list is at least as large as the largest element from the guarding list. After this, the element y is removed from the guarding list. This is repeated until the guarding list is empty. The answer is then located in the candidate list.

A short pseudo-code description of the APPT algorithm is as follow:

```
1. Initiate empty sorted candidate list, $C$, of length $k$.

2. Initiate empty sorted guard list, $G$.

3. For each block, $B$, covered by the query region (totally or partially) do:

    (a) Insert max$\{k, r\}$ of the largest elements from the block into $C$.
    (b) Insert the $r^{\text{th}}$ largest element from $B$ into $G$.

   For each element $y$ in $G$ do:

    (a) if $y > x$ or $\|G\| < k$ then
        i. For each element, $e$, in the data cube corresponding to the block of $y$
           do: If $e > x$ then insert $e$ into $C$.
    (b) Remove $y$ from $G$.

4. The answer in now in $C$.
```

In the above description, we have left out the details of searching, inserting and deleting elements of the lists and of using the overflow array. Some remark needs to be made on the use of the overflow array. In the original paper [LLAL02], the cache is of fixed size. Whenever the data cube is scanned for candidates, the elements visited are considered for the overflow array (Notice that these are elements of rank $r+1, r+2, \ldots$ and so on). A heuristic method has been designed to determine which elements will be inserted into the overflow array. We are interested in worst-case performance and the use of the overflow array does not result in fast worst-case performance of the query algorithm described. Nevertheless, the extra memory of the overflow array does improve the average performance.

In Figure 4.1, a two-dimensional example of the LPC is shown. In this example, the block size is 4 and $r = 3$, so the three largest elements from each block of the data cube are stored in the LPC. The thin lines represents block boundaries. Observe that the elements in each block of the LPC are stored in sorted order. The thick line represents the query region. The shaded areas are elements considered by the query algorithm as candidates. Note that, besides the information in the figure, the indices

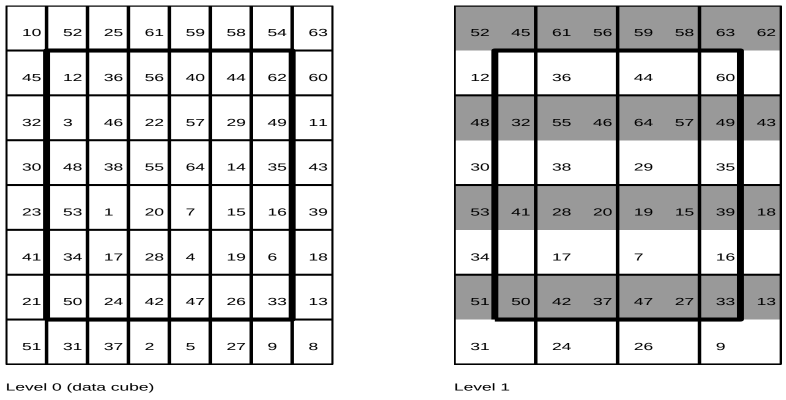

Figure 4.1: Data cube and Location Pre-Computed Cube (LPC) structure with query. Shaded areas are elements considered by the query.

for each element are also stored in the blocks of the LPC. The algorithm considers all blocks that are completely or partially covered by the query region which, in this case, is all blocks of the LPC. For each element, the algorithm uses the index of the element in order to decide if the element are located inside the query region or not.

Although the above algorithm performs well under some statistics assumptions, as shown in [LLAL02], it runs in a time proportional to that of the straightforward method mentioned in the beginning of this section in the worst case, even when $k \leq r$.

Proposition 1. *The APPT algorithm takes $\Theta(m_1 \times m_2 \times \cdots \times m_d)$ time in answering a range top-k query, $Q\,(k; a_1, b_1; \cdots; a_d, b_d)$, in the worst case, where $m_i = b_i - a_i$ for $i = 1, 2, \cdots, d$.*

Proof. Consider first the case when $k > r$. In this case, the APPT method may have to visit all the elements in the query region while maintaining the invariant. This is because if $k > r$, still only the r largest elements in each block is stored. When computing the elements less than the r^{th} largest in one block, all elements of that block need to be visited. To be precise, the number of elements visited is at least $p^d - r$ for each block, where p is a constant denoting the size of the partition interval in each dimension. Therefore, the total number of elements accessed is $\Theta(m_1 \times m_2 \times \cdots \times m_d)$. The query time is then $\Theta(n^d)$ when $m_i = \Theta(n)$ for all $1 \leq i \leq d$.

For the case when $k \leq r$, the query can be answered slightly faster, but with the same order of magnitude. Notice that when the candidate list is constructed, the APPT method will compute the k largest values among all elements stored from each block covered (totally or partially) by the query range. Since the number of blocks under consideration is at least

$$\frac{m_1 \times m_2 \times \cdots \times m_d}{p^d} ,$$

the time required for the construction of the candidate list is

$$k\frac{m_1 \times m_2 \times \cdots \times m_d}{p^d} .$$

Note that p is a constant. By letting $m_i = \Theta(n)$, for $i = 1, 2, \ldots, d$, building the candidate list takes $\Theta(n^d)$ time. □

Although the APPT method has the same complexity as that of the straightforward method, the hidden constant in the time complexity is often much lower than that of the latter one. At the same time, the APPT method does have good average case performance as shown by Loh et al. [LLAL02] through experimental test of real data. Nevertheless, from the above proposition, when the query range is large, the APPT method can result in a big time delay in responding range top-k queries. In the next section, we will propose a new algorithm for the range top-k query problem with better worst case performances which are, therefore, better suited for large queries.

4.2 Hierarchical Blocking

We will show in this section how to reduce the worst-case query cost significantly by employing a hierarchical blocking with ℓ levels on data cubes.

Given a d-dimensional data cube of size n in each dimension, we partition the data cube into $\frac{n^d}{p^{dj}}$ sub-cubes on the j^{th} level of the hierarchy, $0 \leq j \leq \ell$, where each sub-cube is a d-dimensional data cube and p is some positive, constant, integer. That is, each dimension of the given data cube is partitioned into n/p^j intervals on the j^{th} level. The intervals from all dimensions at the j^{th} level form d-dimensional blocks each of size p^{dj}; called the partition blocks. The partition blocks at level 0 corresponds to the data cube itself. For each partition block, the r largest elements in the corresponding range of the data cube are computed and stored. Observe that the blocks at a higher level (larger j) completely include the blocks at the lower levels (smaller j). That is, blocks at higher levels store the r largest elements from the blocks at lower levels.

4.2.1 Query

A d-dimensional range top-k query, $Q\left(k; a_1, b_1; a_2, b_2; \ldots; a_d, b_d\right)$, consists of a query range (a sub-cube of the given data cube) specified by an interval $[a_i, b_i]$ on each dimension i $(1 \leq i \leq d)$ and the value of k. Denote by m_i the size of the query interval along dimension i (that is, $m_i = b_i - a_i + 1$). It is clear that for any i

($1 \leq i \leq d$), there exists some j ($0 \leq j \leq \ell$) such that $p^{j-1} < m_i \leq p^j$, where $p^{-1} \equiv -\infty$.

The algorithm uses a set of *candidates*, for the top-k elements of the query region. For every range top-k query, the partition blocks covered by the query region (totally or partially) can be classified into two categories: The *interior blocks* (A, in Figure 4.2) and the *border blocks* (B and C, in Figure 4.2). The interior blocks are those completely included in the query region while the border blocks are the blocks partially included in the region. We call the region with interior blocks, the *interior region* and the region with border blocks the *border region*.

Our algorithm first finds hierarchical level i such that this is the highest possible level where some partition blocks in the query range becomes interior blocks. This is the level with the largest blocks. Depending on whether the parameter k is bounded or not, we have the following two cases:

Case $k \leq r$

Consider the case when there are more than k largest elements stored for each block. After finding the highest hierarchical level, i, the algorithm computes the k largest elements from all the elements stored for the interior blocks at level i and inserts them into the candidate list. Next, a recursive call of the algorithm is performed on each of the border regions. By combining the answers from the border blocks with the candidates from the interior blocks, we obtain the answer to the original top-k range query. Since $k \leq r$, there is no need to further investigate the interior blocks at levels lower than i. Since we have stored r largest elements in each block, we can get the answers from each block in at most k steps. The elements stored in the interior blocks becomes candidate elements. Together with the relevant elements from the border region, those elements will form the answer to the original query. The candidates from the interior region together with the answers from recursive applications of the algorithm on the border regions will give the candidates for the answer to the original query.

On handling the border blocks, the algorithm does not use the information stored from the actual border blocks, because they are only partially included in the query region. Instead, the algorithm recurses down one level at a time along the hierarchical levels on the area formed by these border blocks. This continues until some blocks are totally covered by this new border query. Those blocks are now the interior blocks for the border query. Note that this will always happen, because blocks are smaller at lower levels. When this happens, the algorithm continues the recursion on the new (thinner) border region. This carries on until there is no border blocks to process. The last, and thinnest, border could, in the worst case, be the data cube itself. The algorithm is recursively applied on each of the border regions (B, in Figure 4.2) and each of the corners (C, in Figure 4.2). Note that the recursion will only continue down the hierarchy on the outer side of the border regions (the side not located towards the inner region). This is because the blocks at lower levels are perfectly aligned with

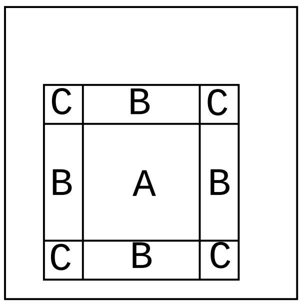

Figure 4.2: A two-dimensional data cube and a query region, with inner region (A) border region (B and C) marked.

the border between the inner region and the border region.

At every recursion stage, the k largest elements of the candidates are returned as answer to the query. In fact, it is not necessary to, at any stage, maintain more than k candidates. However, we choose not to present it for simplicity. Observe that the indices of the elements stored for each block are not used by the algorithm and thus need not to be stored. This will decrease the space usage by a factor of d, for the d-dimensional case.

In Figure 4.3, a two-dimensional example of our algorithm is shown, where $r = 3$ (three largest elements per block). Observe that the elements stored in each block are sorted in non-increasing order. The thin lines represents block boundaries in the data structure. The thick line represents the query region for this particular query. Shaded areas are elements considered as candidates by the query algorithm. The interior blocks of this query consists of the four blocks with shaded regions in level 1. The represent the interior region since they are completely covered by the query region. There is no other such block at level 1. In level 2, there is no blocks that are completely covered by the query region and, therefore, no elements from this level is considered as candidates for this query. Note that in this simple example, it appears that the interior region will almost never consist of blocks from level 2. This is because the value of p (the side-length of the blocks) is as low as 2 in this example. If p would have been larger, there would have been more blocks at level 2, and the situation would be different. Since there are no more blocks at level 2 that is completely covered by the query region, the algorithm recurses to level 0, in order to find blocks that are covered. Hence, all border blocks consists of blocks from level 0. The candidates from those blocks are all considered. The answer to the query is the k largest among all those values considered as candidates.

The pseudo-code description of our algorithm is as follows:

- Create an empty list, L, of length k. Maintain this list sorted.

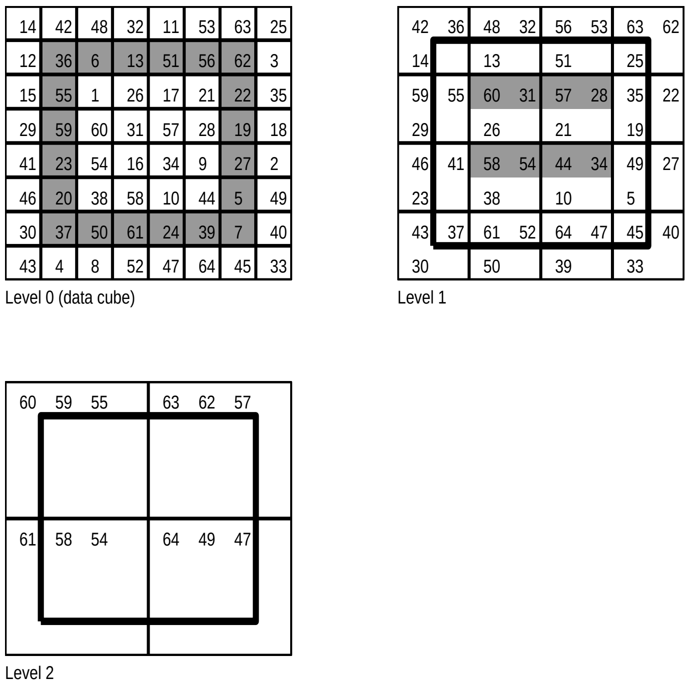

Figure 4.3: Hierarchical structure with 3 levels and $r = 3$. Query region is marked with thick lines. Shaded areas are elements considered as candidates.

- Find the highest level (largest i) such that at least one block is completely inside the query region. Compute the k largest elements from those blocks and insert into L.
- For each region covered by the border blocks, do the following:
  - Call algorithm recursively on all border blocks (B and C, in Figure 4.2). Stop the recursion at the lowest level (that is, the data cube itself).
  - Add the answer (k elements) from the recursive calls to L.

The above algorithm has a good worst-case performance due to the fact that it has a mechanism that selects the candidates from interior blocks fast. In particular, the algorithm runs in linear time for two-dimensional range top-k queries. First, we will need the following observations: Clearly, for a given d-dimensional range top-k query $Q(k; a_1, b_1; a_2, b_2; \dots; a_d, b_d)$, the number of interior blocks at level i ($0 \leq i \leq \ell$) with respect to the query Q is bounded by a number linear in $m_1 + m_2 + \dots + m_d$, where

$m_j = b_j - a_j + 1$ for $j = 1, 2, \ldots, d$. Take the two dimensional case as a n example. Assume without loss of generality that $m_1 \geq m_2$. Consider first two extreme cases. If $m_1 = m_2$, then the number of interior blocks is at most $p^d = p^2$ (here, $d = 2$). On the other hand, if $m_1 = \Theta(m_2)$m the number of interior blocks increases but are bounded by some constant. Implicitly, we assign the number of hierarchical levels, l, to be $\Theta(\log n)$. Another extreme case is when $m_2 = 1$, In this case, the number of interior blocks with respect to the query Q is at most m_1. While the value of $m_2 = 2p$ the number of interior blocks is at most $2\frac{m_1}{p}$. If the value of m_2 is between 1 and $2\frac{m_1}{p}$ the number of interior blocks is at most $2pm_1$. This way of reasoning can be applied to all levels. It is important to notice that the number of interior blocks with respect to the query, Q, decreases from m_1 when the value of m_2 increases. Therefore, the number of interior blocks is $O(m_1 + m_2)$ (not $O(m_1 \times m_2)$. More precisely,

Observation 3. *The number of blocks at level i ($0 \leq i \leq l$) with respect to any range top-k query $Q(k; a_1, b_1; a_2, b_2; \cdots; a_d, b_d)$ is at most*

$$O\left(d \max_{1 \leq j \leq d} \frac{m_j}{p^i}\right) ,$$

where $m_j = b_j - a_j + 1$ and p is the side-length of the partition block.

Since we are only considering the case when $k \leq r$, we need only to access the interior blocks at the highest possible level, i, because of the fact that all the precomputed values of these blocks are the only candidates to the query. Recall that for each interior block at level i, we have already precomputed and stored the r largest elements. Thus, the elements in the candidate list form an answer to the query, Q, restricted to the interior region. Constructing the candidate list costs the same as computing the largest k elements among a set of m sorted sequences each of length $\geq k$, which is at most $\Theta(mk)$. Therefore,

Observation 4. *The time consumed by our query algorithm on the interior blocks is at most*

$$O\left(kd \max_{1 \leq j \leq d} \frac{m_j}{p^i}\right) ,$$

which equals $O(n)$ if $m_j = \Theta(n)$ for $1 \leq j \leq d$ and k is some constant less than r.

Our algorithm is extremely fast for the two dimensional case. The reason for this is that the number of partition blocks accessed when recursing on the border region is bounded linearly.

Theorem 9. *Given a two-dimensional data cube with size n in each dimension, a range top-k query $Q\,(k; a_1, b_1; a_2, b_2)$ on the cube can be answered in $O(m_1 + m_2) = O(n)$ time in the worst case , where $m_i = b_i - a_i + 1$ for $i = 1, 2$, if $k \leq r$ and r is constant.*

Proof. Notice first that the query algorithm accesses only the blocks at the highest possible level (that is, the largest possible interior blocks) and is invoked recursively on the border region. From Observation 4 we know that at most $O(O(m_1 + m_2)k) = O(m_1 + m_2)$ time is needed to process all the interior blocks.

For the recursion on the border region, consider the border region (B, in Figure 4.2). Now, the number of blocks processed, in the worst case, at the lowest level (the data cube) is m_i. The number of blocks processed at the level above the data cube is

$$\frac{m_i}{p} \ .$$

In general the number of blocks processed decreases by a constant factor for each level. Therefore, in the worst case, the number of processed blocks at all the levels above the data cube are less than the number of elements processed from the data cube. Hence, the processing time is dominated by the data cube level, in the worst case, which is $O(m_i)$ for dimension i.

Similarly, the corner border region (C, in Figure 4.2) requires $O(m_1 + m_2)$ time to process.

The total number of blocks accessed by the query algorithm is therefore $O(m_1+$ $+m_2)$. The time required to process each block is constant, since r is constant. The theorem follows. $\square$

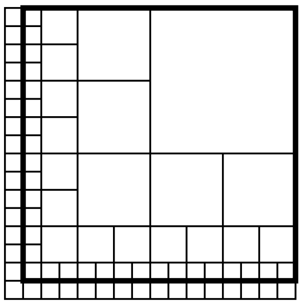

Figure 4.4: Recursion while performing a query. Smaller blocks are used in the border region. The thick line represents the query region.

In the above proof, because the parameter r is constant, the complexity will be of the same order as the number of blocks accessed in the data structure. However, if we choose to parameterize the complexity in r and k (that is, they are not constants), there are two possible ways to store theses r elements, that need to be considered: Either the r elements are stored in non-decreasing order for each block or they are not sorted.

Consider the first case of unsorted elements. When processing a single block, the selection of the k^{th} largest element, x, of a single block could be done using a linear-time selection algorithm [CLR90]. Then, the all elements larger than x could be found by traversing all the r elements and choosing the elements that are larger than x. If we maintain a list of k elements, we could, for each new processed block, combine the k elements from that block with the elements from the list with a similar method. This would require $O(r)$ steps per processed block. Hence, the total time required for a query would be $O(r(m_1 + m_2))$.

Now, consider case of sorted elements in each block. For a single block, we could perform the selection of the k largest elements in merely k steps, since the elements in the block are sorted. The k selected elements would also be sorted. We process one block at a time and maintain only k candidates. It is easy to merge the k new candidates from each processed block with the current k candidates in $O(k)$ steps. In this case, we get a total processing time for a query of $O(k(m_1 + m_2))$.

Our algorithm runs in linear time if the number of pre-computed aggregation values is not less than the parameter value k in the range top-k query. This is, of course, very efficient. In practical real data settings, one may assume that all range top-k queries ask for a value of k bounded above by the maximum value k_{max}, as the case when investigating the complexity for ranking aggregates in [LYAA04]. In this case, we can choose the parameter r of our algorithm to be k_{max} and achieve the linear time solution.

Case $k > r$

If $k > r$, we can modify the above algorithm so that the performance will be as good as that of the APPT. Similarly, we start at the highest level i in the hierarchical structure as the above algorithm. Then, depending on the performance requirement, we continue the procedure in either of the following strategies. If we are interested in a solution fast in the worst case, we apply the algorithm directly now to the lowest level (that is, the data cube level) for each and every block. On the other hand, if the average case complexity is the focus, interior blocks will then be processed in a way similar to that of the APPT beginning from the level i downwards. The APPT will access the data cube if not sufficiently many candidates are found in the structure. Here, we can recursively access lower levels of the structure, until enough elements are found. This will give the same asymptotic performance as for the APPT. In general, the time complexity will be the same asymptotically as the performance of the APPT when $k > r$. For the case of $k \leq r$, our algorithm results in a significant improvement over the previously known algorithms.

4.2.2 Updates

In this subsection, we will discuss the update cost of our method in a dynamic data cube. When a value in the data cube is changed, all the partition blocks corresponding to the updated cell need to be checked for the element. For a hierarchical structure of ℓ levels, there are ℓ such blocks. If the old value of the updated cell is found in the pre-computed list of the r largest elements of the block, this old value will be removed from the list. If the new value of the updated cell is large enough to qualify for the list, it will be inserted into the list. If not, another element (not in the list) has to be found from the blocks at lower level (or even at the data cube level). This can be achieved by performing a range top-r query for the block in question. The smallest element of the answer from this range top-r query is then added to the list of the block. It may be the case that we need to perform such a range query for each level. The blocks are processed in a bottom-up fashion with respect to level (where the data cube is the lowest level). Now, we note that, the query process needs not descend more than one level down, for each block, because the element searched for will always be found one level down. This is because we processed that level before and it is, therefore, updated with the new element. Formally, the update procedure is as follows.

1. Update the cell in the data cube with the new value, called the new value x. Store the old value, y, of the updated cell.
2. Let $i = 1$.
3. While $i \leq \ell$ do for the block, B, corresponding to the updated cell at level i:
   (a) If y is in the top-r list of the block B, then
      - remove y from the list. (The list now contains $r - 1$ elements);

      else
      - goto Step 3d.

   (b) If the new value x is large enough to qualify for the list of B, then
      - insert the x into the list.
      - We are now done with this level: goto Step 3d.

   (c) Find the r largest elements from the lists of the blocks at level $i - 1$ (the lower level) and replace the list of B by these elements.
   (d) Let $i \leftarrow i + 1$.

Notice that the top-r list of one block at each level of the hierarchy structure has been updated. There are ℓ levels in the structure. For each updated block at level i, all blocks covering the same area as the updated block at level $i + 1$ needs to be accessed. There are p^2 such blocks. For each such block, r elements are accessed. Hence, the total update cost will be $\Theta(r\ell p^2)$. Letting $\ell = \log n$ and noting that p^2 is a constant, we have

Theorem 10. *The update in the hierarchy structure of our algorithm takes $\Theta(r \log n)$ time in a given d-dimensional data cube of size n in each dimension in the worst case.*

4.3 Higher Dimensions

When applying the blocking hierarchical algorithm to solve higher (> 2) dimensional range top-k problems, the worst-case query complexity will still be at least as good as that of the APPT. For some large queries, the performance will be better. However, the complexity will not be linear, as in the two-dimensional case. This is because the worst number of blocks needed to be processed is super-linear.

4.4 Conclusions

We have design an efficient algorithm for solving the range top-k problem that has a linear time worst-case performance for a 2-dimensional data cube. The performance in higher dimensions also improve over previous known methods. Event hough our approach is quite straightforward, our method do perform well in the worst-case. The linear time complexity is achievable for the case when all the range top-k queries request for the parameter k that is bounded above by a maximum value. This is the most used case in practice. New ideas will be needed for the case when the parameter k is not unbounded. Moreover, experimental results of our algorithm on some real data would be interesting as well.

Chapter 5

Conclusions

We have designed several efficient algorithms operating on the data cube and provided some new insight into the problems.

Considering the new algorithms presented and discussed in this thesis, several interesting research questions arises. For the k *Maximum Sum Subsequences* problem, we note that a trivial lower bound for the problem is $\Omega(n+k)$, where n is the number of input elements and k is the number of subsequences requested. Our algorithm is optimal for a value of the parameter $k \geq n\log^2 n$ and $k = \Theta(1)$. For the remaining values $\Theta(1) < k < n\log^2 n$, our algorithm is not matched by the lower bound. We believe that an improvement of the algorithm removing the square on the logarithmic term should be possible.

Notice that in the computation of the rank of the element x, there is a lot of correlation between computations. At one stage, x is doubled between each iteration. It might be possible to utilize this fact in order to speed-up the algorithm. Moreover, in each iteration, several searches for the same element x is performed in data that is not uncorrelated.

Considering the trivial lower bound again, if this lower bound is not tight, it would be interesting to develop a tight lower bound and a matching, possibly improved, algorithm.

Similarly, considering the *range-sum* data structure and algorithm, an interesting question is whether the query-update time trade-off provided by our algorithm is optimal. Our belief is that it is not possible to improve the trade-off using a similar approach for the solution. New ideas would be necessary. Another interesting question is the memory usage of our, and many other, algorithms. It is important to note that the magnitude of the elements in our data structure are slightly larger than the corresponding magnitude of the elements in the original data cube. One may argue that this is just another way of using more memory. However, the approach used in our algorithm is well accepted by the research community and reasonable, even in practice. Nevertheless, it is important to raise the question of how large the magnitude of elements is allowed to be. In theory (and even practice) it is very possible to store arbitrary much information in one single element, if the magnitude of the element is allowed to grow arbitrary large. Of course, this is where the computational model ceases to be implementable; it is not possible to perform constant-time operations on an arbitrary large element. In many algorithms, the sum of several element is aggregated in one element which results in elements slightly larger than the original elements. This is accepted practice, but can one aggregate the product of equally many elements? The elements in such a data structure would probably be much larger than most people can accept. Because of this, it is always important to, at least to some extent, consider the possibility of implementation when developing data structures and algorithms, at least if they are going to be implemented efficiently.

For the *Range top-k* problem, similar questions arise. Since this data structure use additional memory to achieve its performance, one might ask whether it is possible to achieve the same performance using less memory. A trade-off between query and update performance is also an interesting issue.

Bibliography

[AG91] S.G. Akl and G.R. Guenther. Application of broadcasting with selective reduction to the maximal sum subsegment problem. *International Journal of High Speed Computing*, 3:107–119, 1991.

[BC04a] Fredrik Bengtsson and Jingsen Chen. Efficient algorithms for k maximum sums. In *Proceedings of The 15th International Symposium on Algorithms and Computation (ISAAC)*, volume 3341 of *Lecture Notes in Computer Science*, pages 135–146, Hkust, Hong-Kong, December 2004. Springer.

[BC04b] Fredrik Bengtsson and Jingsen Chen. Space-efficient range-sum queries in OLAP. In *Proceedings of 6th International Conference on Data Warehousing and Knowledge Discovery*, volume 3181 of *Lecture Notes in Computer Science*, pages 87–96, Zaragoza, Spain, September 2004. Springer.

[BE97] Ramon Barquin and Herb Edelstein, editors. *Building, Using, and Managing the Data Warehouse.* The Data Warehousing Institute Series. Prentice Hall PTR, Upper Saddle River, New Jersey 07458, 1997.

[Ben85a] J.L. Bentley. Programming pearls: Algorithm design techniques. *Communications of the ACM*, 27:865–871, 1985.

[Ben85b] J.L. Bentley. Programming pearls: Perspective on performance. *Communications of the ACM*, 27:1087–1092, 1985.

[BT04] Sung Eun Bae and Tadao Takaoka. Algorithms for the problem of k maximum sums and a VLSI algorithm for the k maximum subarrays problem. In *Proceedings of the 7th International Symposium on Parallel Architectures, Algorithms and Networks*, pages 247–253, 2004.

[CCLL01] Seok-Ju Chun, Cin-Wan Chung, Ju-Hong Lee, and Seok-Lyong Lee. Dynamic update cube for range-sum queries. In *Proceedings of the 27th VLDB Conference*, pages 521–530, 2001.

[CCLL02] Seok-Ju Chun, Cin-Wan Chung, Ju-Hong Lee, and Seok-Lyong Lee. Space-efficient cubes for OLAP range-sum queries. Technical report, Korean Advanced Institute of Science and Technology, January 2002.

[CG99] Surajit Chandhuri and Luis Gravano. Evaluating top-k selection queries. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 397–410, 1999.

[CKA96] Martin Campbell-Kelly and William Aspray. *Computer*. BasicBooks, 10 East 53rd Street, New York, NY 10022, first edition, 1996.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts, first edition, 1990.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[cod71] CODASYL database task group report, April 1971.

[cod78] CODASYL data description language committee, journal of development, January 1978.

[Cod95] E. F. Codd. A relational model of data for large shared data banks. *ACM Classic of the month*, November 1995.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progression. *Journal of Symbolic Computation*, 9:251–280, 1990.

[DR99] Donko Donjerkovic and Raghu Ramakrishnan. Probabilistic optimization of top n queries. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 411–422, 1999.

[EN00] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Adison Wesley, third edition, 2000.

[EN04] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Adison Wesley, fourth, international edition, 2004.

[FJ82] G.N. Frederickson and D.B. Johnson. The complexity of selection and ranking in $X+Y$ and matrices with sorted columns. *Journal of Computer and System Sciences*, 24:197–208, 1982.

[FJ84] G.N. Frederickson and D.B. Johnson. Generalized selection and ranking: Sorted matrices. *SIAM Journal on Computing*, 13:14–30, 1984.

[GAAS99] Steven P. Geffner, Divyakant Agrawal, Amr El Abbadi, and T. Smith. Relatve prefix sums: An efficient approach for querying dynamic OLAP data cubes. In *Proceedings of the 15th International Conference on Data Engineering*, pages 328–335, 1999.

[GRAA99] Steven P. Geffner, Mirek Riedewald, Divyakant Agrawal, and Amr El Abbadi. Data cubes in dynamic environments. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, pages 31–40, 1999.

[Gre78] U. Grenander. *Pattern Analysis.* Springer-Verlag, New York, 1978.

[Gri82] D. Gries. A note on the standard strategy for developing loop invariants and loops. *Science of Computer Programming*, 2:207–214, 1982.

[HAMS97] C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 73–88, 1997.

[KLMHL98] D. W. Kim, E. J. Lee, Kim M. H, and Y. J. Lee. An efficient processing of range-min/max queries over data cube. *Information Sciences*, 112:223–237, dec 1998.

[KMS03] Danny Krizanc, Pat Morin, and Michiel Smid. Range mode annd range median queries on lists and trees. In T. Ibaraki, N. Katoh, and H. Ono, editors, *Algorithms and Computation*, pages 517–526, 2003.

[LLA$^+$01] Z. H. Luo, T. W. Ling, C. H. Ang, S. Y. Lee, and B. Cui. Range top/bottom k queries in OLAP sparse data cubes. In *Proceedings of 11th International Conference on Database and Expert Systems Applications (DEXA)*, volume 2113 of *Lecture Notes in Computer Science*, pages 678–687, 2001.

[LLAL02] Zhen Xuan Loh, Tok Wang Ling, Chuan Heng Ang, and Sin Yeung Lee. Adaptive method for range top-k queries in OLAP data cubes. In *Proceedings of 13th International Conference on Database and Expert Systems Applications (DEXA)*, volume 2453 of *Lecture Notes in Computer Science*, pages 648–657, 2002.

[LLL00a] Sin Yeung Lee, Tok Wang Ling, and HuaGang Li. Hierarchical compact cube for range-max queries. In *Proceedings of the 26th Internal Conference on Very Large Databases*, pages 232–241, 2000.

[LLL00b] Hua-Gang Li, Tok Wang Ling, and Sin Yeung Lee. Range-max/min query in olap data cube. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications*, number 1873 in LNCS, pages 467–476, 2000.

[Loo80] Mary E.S̃. Loomis. The 78 codasyl database model: a comparison with preceding specifications. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 30–44, 1980.

[LYAA04] Hua-Gang Li, Hailing Yu, Divyakant Agrawal, and Amr El Abbadi. Ranking aggregates. Technical Report 2004-07, University of Calofornia at Santa Barbara, Computer Science department, 2004.

[Maa94] Wolfgang Maass. Efficient agnostic pac-learning with simple hypotheses. In *Proceedings of the Seventh Annual Conference on Computational Learning Theory*, pages 67–75, 1994.

[mCL03] Kai min Chung and Hseueh-I Lu. An optimal algorithm for the maximum-density segment problem. In *Algorithms - ESA 2003, 11th Annual European Symposium*, volume 2832 of *Lecture Notes in Computer Science*, pages 136–147. Springer, 2003.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[PD95] K. Perumalla and N. Deo. Parallel algorithms for maximum subsequence and maximum subarray. *Parallel Processing Letters*, 5:367–373, 1995.

[Poo01] Chung Keung Poon. Orthogonal range queries in OLAP. In *Proceedings of the 8th Internatioal Conference on Database Theory (2001)*, pages 361–374, 2001.

[QA99] K. Qiu and S.G. Akl. Parallel maximum sum algorithms on interconnection networks. Technical Report No. 99-431, Jodrey School of Computer Science, Acadia University, Canada, 1999.

[RAA01] Mirek Riedewald, Divyakant Agrawal, and Amr El Abbadi. Flexible data cubes for online aggregation. In *Database Theory - ICDT 2001, 8th International Conference, London , UK, January 4-6, 2001, Proceedings*, volume 1973 of *Lecture Notes in Computer Science*, pages 159–173, 2001.

[RAAP00] Mirek Riedewald, Divyakant Agrawal, Amr El Abbadi, and Renato Pajarola. Space-efficient data cubes for dynamic environments. In *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery (DaWak)*, pages 24–33, 2000.

[Ran73] Brian Randell, editor. *The Origins of Digital Computers Selected Papers.* Springer-Verlag, Berlin, Hidelberg, New York, 1973.

[Smi87] D.R. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, 8:213–229, 1987.

[Tak02] T. Takaoka. Efficient algorithms for the maximum subarray problem by distance matrix multiplication. In *Proceedings of the 2002 Australian Theory Symposium*, pages 189–198, 2002.

[TT98] H. Tamaki and T. Tokuyama. Algorithms for the maximum subarray problem based on matrix multiplication. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 446–452, 1998.