NTNU
Norwegian University of
Science and Technology

Efficient spatio-interval join

Eirik Rognø

Master of Science in Computer Science
Submission date: June 2018
Supervisor: Kjetil Nørvåg, IDI

Norwegian University of Science and Technology
Department of Computer Science

Efficient spatio-interval join

Eirik Rognø

June 11, 2018

Problem description

With the trend of increasing amounts of spatio-interval data, the goal of the this thesis is to investigate efficient algorithms for joining datasets on spatio-interval properties. The task is to study existing techniques for spatial and interval joins, development of new algorithms and experimental evaluation.

Supervisor: Kjetil Nørvåg

Abstract

This thesis studies existing techniques for both spatial join and interval join. Using existing methods and algorithms for join operations two new algorithms for spatio-interval join is developed. Parallelization techniques from work on spatial join and interval join are applied to the two spatio-interval join algorithms and a parallel algorithm is presented for each of them. The results from implementing these algorithms show that it is possible to solve the spatio-interval by modifying the existing algorithms and that the parallelization techniques can be successfully applied to the spatio-interval join algorithms.

Sammendrag

Denne oppgaven undersøker eksisterende teknikker for å utføre join operasjoner for rom og intervall. Ved bruk av eksisterende metoder og algoritmer blir to nye algoritmer for join av data på både rom og intervall samtiding utviklet. Parallelliseringsteknikker fra arbeider med intervall join og romlig join blir så benyttet på de to algoritmene til å lage to parallelle algoritmer for å utføre join på rom og intervall samtidig. Resultatene fra implementasjonen av disse algoritmene viser at det er mulig å utføre en join operasjon på både rom og intervall ved å modifisere de eksisterende algoritmene. I tillegg ser vi at parallelliseringsteknikkene for de eksisterende algoritmene for intervall join og romlig join kan brukes på de nye algoritmene for både romlig og intervall join.

Contents

Chapter 1

Introduction

Spatio-interval data is data containing both a spatial area as well as an interval of some kind. This data can be created in many different scenarios. If you have the GPS tracker in your smart-phone enabled it generates spatio-interval data about your geographical position at all times. Weather data is another area which contain a lot of spatio-interval data. Measurements of weather events like rainfall, temperature and pressure will all include a spatial component as well as an interval in time [5]. With the rise of the internet of things more and more internet connected sensors exist, these can all potentially create spatio-interval data.

This thesis tries to solve the problem of a spatio-interval join. That is a join of data where both a spatial component and an interval component overlaps. In the case of data generated from a smartphone a spatio-interval join could for example be used to find which phones where at the same location during a given interval in time.

Several methods and algorithms for conducting both interval and spatial joins already exists today. One of the more common approaches for spatial join in main memory is the plane sweep method [10] which will be discussed in section 5.2. For interval join one method is the endpoint-based interval join proposed in [3]. The endpoint-based interval join will be discussed in section 5.1. Several database systems already support interval and spatial operations. Examples of this are SQL:2011 [11] and SpatialHadoop [8]. These algorithm will be expanded upon in this thesis. New algorithms for spatio-interval join will be created by modifying and adding to the existing algorithms.

The algorithms discussed here are main memory algorithms. The growing memory sizes in modern computers along with the increasingly lower price of memory has enabled larger data to be processed in main memory, and most modern database vendors already provide an in-memory solution [12]. To fully utilize the computing power of modern processors, it is necessary to design parallel algorithms. In this thesis two different parallel algorithms for solving the

spatio-interval join are presented.

Using the existing algorithms for spatial and interval join, this thesis will describe some new algorithms for spatio-interval join. The thesis starts with an overview of some related works in chapter 2, before explaining some key concepts and definitions as well as a more in-depth explanation of the spatial and interval algorithms used in chapter 3. In chapter 4 an overview of some key concepts in computer architecture and parallel processing are described. In chapter 5 the spatial join and interval algorithms used in the thesis are explained. Chapter 6 discusses some possible optimizations for the algorithms described in chapter 5. In chapter 7 several sequential algorithms and methods for spatio-interval join is described, while chapter 8 shows how this algorithms can be parallelized. Chapter 9 describes how the algorithms where implemented and chapter 10 presents and discusses some experimental results. The final chapter outlines some possible future work in the area of spatio-interval join.

Chapter 2

Related works

A lot of work has been done on both spatial join algorithms and interval join algorithms, and some existing work also focus on the area of spatio-interval operations. For interval joins this thesis mainly focuses on the endpoint-based index join described in [15] and section 5.1, however several other methods for interval joins exist. One method is the overlap interval partition join described in [6]. In a partition join the domain of the input collections are divided into partitions. The overlap interval partition join algorithm first divides the domain into equally sized granules, then each interval is assigned to the partition that consists of the smallest sequence of granules that contain the interval. After the partitions are created the join algorithm iterates over the partitions of one of the input collections and joins them with the overlapping partitions of the other input collection. In [3] a forward scan based plane-sweep method for interval join is described. The forward scan based plane-sweep first sorts the input collections by the start endpoint of each interval. The algorithm iterates through the sorted collections, stopping at each start point in the input collections. For every interval it encounters the plane-sweep scans forward, outputting a join with all intervals from the other collection that start before the end of the current interval.

In [10] many different methods for conducting spatial joins are described, one of them being the plane-sweep which is discussed more in section 5.2. Another method described is the nested-loop join. This is the most basic method of joining. Given two collections of spatial objects the nested loop join simply compares every element of one collection with every element of the other. An improvement over the nested loop join is the indexed nested loop join. In the indexed variant a spatial index is first built for one of the relations, then the algorithm iterates through the other relation and checks for an intersection by searching the index of the first collection. In [7] seven methods for indexing collections of spatial objects are described and evaluated. These include the R-Tree, the R+-tree, the uniform grid, the quad tree, the K-d tree, the Hilbert curve and the Z-curve.

The R-tree and its variations are a widely used index for spatial objects. In [9] Guttman describes the R-tree as a height-balanced tree similar to the B-tree. It is designed so that a spatial search only needs to visit a small number of nodes. Each leaf node contains a pointer to an object in the database and a rectangle that is the bounding box of the object it indexes. Every non-leaf node contains a pointer to a child node further down the tree and a rectangle that completely encloses all the lower nodes entries. The tree is updated dynamically when entries are added or deleted. Several variations and optimizations of the R-tree exists. One of these is the R*-tree described by Beckmann et al. in [1]. The R*-tree mostly works in the same way as the R-tree, the difference is how it optimizes when inserting and deleting from the tree. The R-tree aims to minimize the area of each enclosing rectangle of the non-leaf nodes while the R*-tree tries to minimize the overlap between the rectangles in different branches of the tree. Additionally the R*-tree introduced the method of reinsertion. Because the R-tree structure is dependent on the order in which elements are inserted the R*-tree will delete and reinsert elements when a node reaches a certain capacity to avoid ending up with a sub-optimal index.

In [18] a method for computing spatio-interval join using R-trees is described. The method described uses a temporal R-tree, the TR-Tree for short. This variation of the R-tree is enhanced to store temporal information as well as the spatial object. Each entry in the tree has two timestamps in addition to the spatial object. The TR-tree stores a collection of R-trees, where each R-tree represents an interval in time [17]. The join is then computed by iterating through one of the indexed relations and traversing the other relation looking for a join similarly to how it is described in [4].

Chapter 3

Preliminaries

R, S	Collections of data. Either spatial data, interval data or spatio-interval data
r, s	Single element from collection
r_i, s_i	Element i from collections R and S
$R \bowtie S$	Join of collections R and S
$r.start, r.end, s.start, s.end$	Start and end of interval associated with objects r and s
$\langle r_i, s_i \rangle$	Output tuple of join

Table 3.1: Table of notations

An interval is defined by a start and an end point in a one dimensional continuous space. The interval join $R \bowtie S$ is defined as all pairs of intervals $r \in R$, $s \in S$ that intersect. Intersection is defined as $r.start \leq s.start \leq r.end$ or $s.start \leq r.start \leq s.end$. Figure 3.1 shows the two interval relations R and S. An interval join between the two relations would result in the pairs:

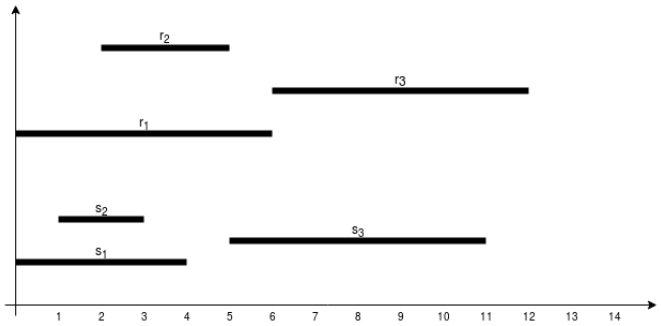

Figure 3.1: Interval relations R and S

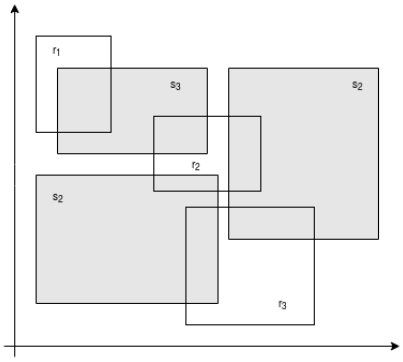

Figure 3.2: Spatial relations R and S

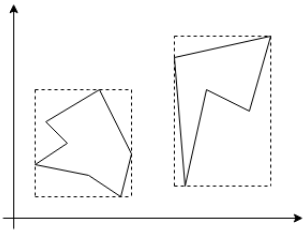

Figure 3.3: Examples of minimum bounding rectangles

$\langle r_1, s_1\rangle, \langle r_1, s_2\rangle, \langle r_1, s_3\rangle, \langle r_2, s_1\rangle, \langle r_2, s_2\rangle, \langle r_3, s_3\rangle$.

For the spatial join the spatial component is defined as some multidimensional object in Euclidean space. The spatial join on this component is defined as an intersection or overlap of two objects. Figure 3.2 show the two spatial relations R and S. Performing a spatial join between R and S would result in the following pairs: $\langle r_1, s_3\rangle, \langle r_2, s_1\rangle, \langle r_2, s_2\rangle, \langle r_2, s_3\rangle, \langle r_3, s_1\rangle, \langle r_3, s_2\rangle$. The objects in this example are all axis-aligned rectangles. In the general case of spatial join the objects can be of all shapes. The spatial join in the general case can be divided into two stages, the filtering stage and the refinement stage [10]. In the filtering stage an approximation of the objects are used, then in the refinement stage the inaccurate results that resulted from the approximation is removed. A commonly used approximation method for two-dimensional objects is the minimum bounding rectangle. MBRs are the smallest rectangle that completely encloses the original objects. Some examples of MBRs are shown in Figure 3.3. In this thesis only the filtering stage of the spatial join is solved, so all spatial objects are assumed to be axis-aligned rectangles.

If R is a spatio-interval relation then for each tuple r_i, $1 \leq i \leq n$, r_i contains both the start and end value of the interval as well as coordinates for the spatial component. In the case where the spatial component is a MBR the tuple contains the coordinates for two of the corners. The spatio-interval join is defined as the intersection of the results from the spatial join and the interval join. If Figure 3.2 describes the spatial component of the relations r and s and 3.1 de-

scribes the interval component then the result from executing the spatio-interval join would be the following pairs: $\langle R_1, S_3\rangle, \langle R_2, S_1\rangle, \langle R_2, S_2\rangle, \langle R_2, S_3\rangle$.

Chapter 4

Computer Architecture

This section gives a brief introduction to the architecture of modern computers, and some of the motivation for developing an in-memory parallel algorithm.

4.1 von Neumann architecture

Modern computer architectures are mostly based on the von Neumann architecture, which was described in [16]. The von Neumann architecture primarily consists of three parts: the control unit, the arithmetic unit and the memory. The control unit is responsible for the logical control of the program. It is responsible for transferring data between the memory and the arithmetic unit as well as sending instructions to the arithmetic unit. The arithmetic unit is responsible for executing the calculations sent to it by the control unit. The main operations for the arithmetic unit is the four basic math operations: addition, subtraction, multiplication and division. Although it can be capable of doing other operations as well. The memory component is responsible for storing all the data needed by the program as well as storing intermediate values transferred by the control unit during execution. Figure 4.1 shows a high-level overview of the architecture.

4.2 Computer memory

In [16] memory is treated like a single component, in modern computer architectures this is not the case. The speed of memory can be a major bottleneck in the system, typically large inexpensive memory runs at a slower speed. To enable the system to perform at high speed while still having large memory capabilities the computer have several forms of memory divided into a hierarchy. Close to the processor is the fastest and smallest memory the process registers

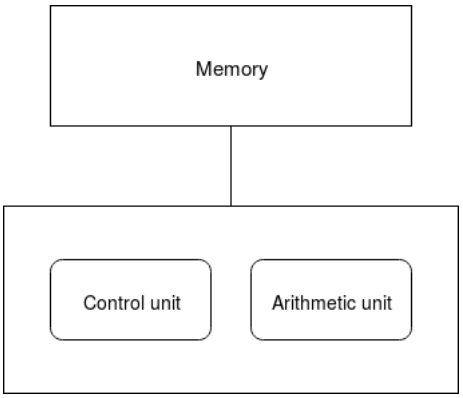

Figure 4.1: von Neumann architecture

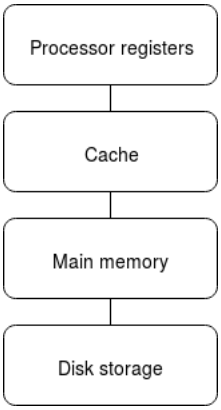

Figure 4.2: The memory hierarchy

as well as the cache. The cache is often subdivided into more than one layer. These are used to hold data already present at the lower levels that are to be used by the processor. After the cache comes the main memory, consisting of RAM. Lastly there is the disk storage, which is the slowest and largest memory. Modern computers normally have a large amount of main-memory. This development has lead to more algorithms being able to assume that all necessary data is transferred to the main-memory first instead of having to optimize for I/O between the disk storage and the main-memory.

4.3 Parallel processing

Due to physical constraints single core processors can not keep increasing their speed as fast as before. To keep increasing the speed of computation it is necessary to utilize parallelism [2]. From 1986 to 2002 the performance of processors increased by 50% every year. After 2002 the progress has slowed down drastically. This has lead to a change in how processors are designed. Instead of

simply creating faster microprocessors most manufacturers now put multiple complete processors on a single circuit. The added processors will not necessarily increase the speed of the serial programs designed to run on one processor. To use the potential of all the processors a program needs to be explicitly written as a parallel program [14].

Chapter 5

Interval and spatial join algorithms

5.1 Endpoint-based interval join

In [15] Piatov et al. describes a method for interval joins called the Endpoint-based interval join. The EBI algorithm starts by creating a data structure called the *endpoint index* for each of the sets in the relation. For each interval r_i, $1 \leq i \leq n$, in an interval relation R the endpoint index contains two entries. Each entry consists of a timestamp, the type of the endpoint and an id. For instance the interval $r_i = \langle r_i.start, r_i.end \rangle$ would create the two entries $\langle r_i.start, start, i \rangle$ and $\langle r_i.end, end, i \rangle$. The endpoints are then sorted by their timestamp, if the timstamps are equal then $start < end$. The EBI algorithm works by scanning the endpoint indices while keeping track of all *active* intervals for each endpoint index. An interval is defined as active if the algorithm has encountered the intervals start-endpoint but not its end-endpoint. When the algorithm encounters a start-endpoint it outputs a join with each active interval of the other relation. Pseudocode for the basic EBI algorithm is given in algorithm 5.1.

5.2 Plane-sweep

One method for executing the filtering stage of a spatial join is the plane-sweep, described in [10]. The plane-sweep algorithm does two passes over the set of rectangles. The first pass sorts all the rectangles in ascending order by their left side. Then it sweeps a vertical scan line through the set of rectangles, stopping each time it encounters a new rectangle at point p. Only the rectangles that intersect the scan line needs to be checked for an intersection with the

Algorithm 5.1 EBI algorithm

Input: Interval relations R and S and endpoint indices E_r and E_s
Output: List of tuples fulfilling the join condition

```
1: active_r ← new Map of tuple identifiers to tuples
2: active_s ← new Map of tuple identifiers to tuples
3: e_r ← first(E_r)
4: e_s ← first(E_s)
5: while exists(e_r) and exists(e_s) do
6:     if e_r < e_s then
7:         if e_r.type = start then
8:             r ← R[e_r.tuple_id]
9:             active_r[e_r.tuple_id] ← r
10:            for all s in active_s do
11:                OUTPUT(r,s)
12:            end for
13:        else
14:            active_r.remove(e_r.tuple_id)
15:        end if
16:        ADVANCE(e_r)
17:    else
18:        if e_s.type = start then
19:            s ← S[e_s.tuple_id]
20:            active_s[e_s.tuple_id] ← s
21:            for all r in active_r do
22:                OUTPUT(r,s)
23:            end for
24:        else
25:            active_s.remove(e_s.tuple_id)
26:        end if
27:        ADVANCE(e_s)
28:    end if
29: end while
```

rectangle associated with p. The rectangles that intersect the scan line are considered *active*. To keep track of the active rectangles a specialized data structure called a *sweep structure* is used. The sweep structure supports three operations, *INSERT*, *SEARCH* and *REMOVE_INACTIVE*. When encountering a new rectangle at point p the *INSERT* operation is used to add the rectangle to the active set. The *REMOVE_INACTIVE* is used to remove all the rectangles that are now completely to the left of the scan line and no longer in the active set. The *SEARCH* operation is used with the current rectangle as input and return all active rectangles that intersect the current rectangle. Pseudocode for the plane-sweep algorithm is given in algorithm 5.2

Algorithm 5.2 Plane-sweep algorithm

Input: Set of rectangles R and S
Output: List of tuples fulfilling the join condition

```
1:  sweep_structure_r ← CREATE_SWEEP_STRUCTURE
2:  sweep_structure_s ← CREATE_SWEEP_STRUCTURE
3:  e_r ← first(E_r)
4:  e_s ← first(E_s)
5:  while exists(e_r) or exists(e_s) do
6:      if e_r < e_s then
7:          sweep_structure_r. INSERT(e_r)
8:          sweep_structure_s. REMOVE_INACTIVE(e_r)
9:          OUTPUT ← sweep_structure_s. SEARCH(e_r)
10:         ADVANCE(e_r)
11:     else
12:         sweep_structure_s. INSERT(e_s)
13:         sweep_structure_r. REMOVE_INACTIVE(e_s)
14:         OUTPUT ← sweep_structure_r. SEARCH(e_s)
15:         ADVANCE(e_s)
16:     end if
17: end while
```

Chapter 6

Optimizations

6.1 Lazy endpoint-based index join

When the EBI algorithm encounters a start endpoint of an interval r_1 from R it scans the active set of the other relation S, $active_s$. If the subsequent endpoints it encounters are the start endpoints of r_2 and r_3, $active_s$ will be scanned to more times, even though it has remained unchanged. To avoid this the algorithm can be modified so that $active_s$ is not scanned until the next endpoint encountered is an endpoint in S, there are no more endpoints in R or the buffer is full. Instead of immediately scanning $active_s$, r is inserted into a buffer. When the buffer is full or if an endpoint form S is encountered $active_s$ is scanned, then for each interval s in $active_s$ the buffer is scanned and it outputs a join with every r in the buffer. Choosing the size of the buffer determines much of the performance gain. Ideally the buffer should fit in CPU cache so that scanning the buffer is much quicker than scanning the active set [15].

6.2 Gapless hash map

The active set in the EBI algorithms needs a data structure that is fast to insert intervals into, remove intervals and scan through. Hash tables are a good alternative for fast insert and removal, as these can be done in time complexity $O(1)$. Unfortunately hash tables are not the most efficient data structure for scanning. To enable the hash map to efficiently scan its elements a new data structure called a gapless hash map is introduced in [15]. The goal of the gapless hash map is to at all times have the elements stored in a continuous area in memory. This is done by storing pointers to the elements in a hash table. Every element then stores a key, a pointer to the next element (if there are more elements in the same bucket), the value and a pointer to the previous element or the hash table index if its the first element in its bucket. Then

each time an element is inserted it is appended to the end of the memory area. Deleting entries are done by first checking if it is the last entry in the storage area. If this is not the case, the entry is swapped with the last one before deleting, and references are updated. This data structure enables the scanning to simply step through the continuous area of memory.

6.3 Linear orderings

A linear ordering is a total order on multidimensional objects, it creates a linear traversal of all objects. One linear ordering is the Z-order. The Z-order divides the domain of the spatial objects into grid cells and each cell is divided into smaller cells. The grid is then traversed in a "Z" pattern and each block at each level of division is fully traversed before moving to the next. In the plane-sweep algorithm described in the previous section the input is sorted by one dimension. For one dimensional objects, like intervals, a one dimensional sort will ensure that neighbouring objects will be next to each other in the ordering. For multidimensional objects this is not the case. Because of this linear orderings play an important role in spatial join techniques, as they will in general keep neighbouring multidimensional objects close together in the ordering [10]. To traverse objects in a Z-order an object can either be assigned to the smallest enclosing cell or an point in the object, for instance the center, can be used to assign the object to a cell. The plane-sweep method can be adapted to use the Z-order instead of sorting the objects by one dimension.

6.4 Sweep structure implementation

How the sweep structure is implemented can have a major impact on the performance of the spatial plane-sweep algorithm. In [10] five alternative data structures for the sweep structure are presented. These are:

- Simple linked list
- Interval tries
- Dynamic segment tree
- Interval tree with a skip list
- Dynamic priority search tree

Excluding the linked list all these have a runtime of $O(log(n))$ for the *search* operation. Where n is the combined size of the two collections being joined. The linked list has runtime of $O(n)$ in the worst case, but in general the sweep structures will contain significantly fewer elements than the entire collection.

Chapter 7

Sequential algorithm

In this chapter two different approaches to the spatio-interval join will be described. One is based on the EBI algorithm discussed in section 5.1, the other is based on the spatial plane-sweep discussed in section 5.2.

7.1 Spatial EBI algorithm

The first spatio-interval join algorithm is created by slightly modifying the EBI algorithm to include a check for an intersection on the spatial component. Most of the algorithm works in the same way as the basic EBI algorithm. The difference is that when the algorithm encounters a start endpoint and loops over the active set to find the intersections an additional check for intersection is done on the spatial component of the objects corresponding to the intervals before adding a spatio-interval intersection to the output. Pseudocode for the spatial EBI algorithm is shown in algorithm 7.1. Since the spatial component of the objects are MBRs a check for an intersection between them can be done in time $O(1)$. Because of this the runtime of the spatial EBI algorithm is the same as the normal EBI algorithm which has a runtime of $O(n^2)$ in the worst case. This is because for every iteration of the algorithm where it encounters a start endpoint it has to scan through the active set of the other collection, which can in the worst case contain n elements. In the average case however the active sets will be much smaller.

7.2 Spatio-interval plane-sweep

Similarly to the EBI algorithm, the spatial plane-sweep can also be modified to solve a spatio-interval join. To check for an interval intersection in the plane-sweep algorithm the sweep structure needs to be modified. Specifically the

Algorithm 7.1 Spatial EBI algorithm

Input: Interval relations R and S and endpoint indices E_r and E_s
Output: List of tuples fulfilling the join condition

```
1: active_r ← new Map of tuple identifiers to tuples
2: active_s ← new Map of tuple identifiers to tuples
3: e_r ← first(E_r)
4: e_s ← first(E_s)
5: while exists(e_r) and exists(e_s) do
6:     if e_r < e_s then
7:         if e_r.type = start then
8:             r ← R[e_r.tuple_id]
9:             active_r[e_r.tuple_id] ← r
10:            for all s in active_s do
11:                if s.MBR overlaps r.MBR then
12:                    OUTPUT(r,s)
13:                end if
14:            end for
15:        else
16:            active_r.remove(e_r.tuple_id)
17:        end if
18:        ADVANCE(e_r)
19:    else
20:        if e_s.type = start then
21:            s ← S[e_s.tuple_id]
22:            active_s[e_s.tuple_id] ← s
23:            for all r in active_r do
24:                if s.MBR overlaps r.MBR then
25:                    OUTPUT(r,s)
26:                end if
27:            end for
28:        else
29:            active_s.remove(e_s.tuple_id)
30:        end if
31:        ADVANCE(e_s)
32:    end if
33: end while
```

SEARCH operation of the sweep structure. The *SEARCH* method is called when encountering new objects. Instead of returning all objects that have intersecting MBRs the *SEARCH* method also checks for an intersection on the interval component of the objects. Similarly to the spatial EBI algorithm discussed in the previous section altering the sweep structure to check for an intersection of the interval component will not give the algorithm a worse runtime since this can be done in $O(1)$ time. As discussed in section 6.4 the runtime of the spatial plane-sweep depends upon the sweep structure implementation. If it is implemented as a linked list the runtime is $O(n^2)$ in the worst case, although it will be lower in practice. Using any of the other implementations the spatial plane-sweep has a runtime of $O(n * log(n))$.

Chapter 8

Parallel algorithm

To make the algorithms more efficient is important to utilize the full capabilities of modern CPUs. As discussed in section 4.3 parallel processing is becoming increasingly more important. This chapter expands upon the sequential algorithms from chapter 7 and describes how they can be parallelized.

8.1 Parallel spatial EBI

In [3] two strategies for parallelizing interval join algorithms are described. The first is a hash-based partitioning algorithm that expands upon the parallelizing technique described in [15]. This partitioning algorithm can also be applied to the spatial EBI algorithm described in section 7.1. The parallelization is achieved by dividing the input R and S into equally sized partitions. The tuples of a relation are sorted by their start time, and the i-th tuple is assigned to a partition using a simple hash-function, i.e. $i \mathrm{mod} k$, where k is the number of partitions. The endpoint indexes are then built separately for each partition. The joining is done pairwise between all partitions of R with all partitions of S. Joining is a disjoint operations so the algorithm can run these independently of each other with no merging in the end. Pseudocode for this approach is given in algorithm 8.1.

The other partitioning method described is a domain-based partitioning algorithm. In the domain-based method the domain of all the interval is partitioned into equally sized tiles. Then for each interval r, let t_{start} be the tile that covers $r.start$ and t_{end} be the tile that covers $r.end$. The interval r i assigned to the partition corresponding to t_{start}, and r is replicated to each of the partitions corresponding to $\{t_{start+1}, t_{start+2}, .., t_{end}\}$. The replicas of r carry a flag to identify them as replicas. After all the intervals of the relations R and S have been assigned and replicated a join is computed for each partition of R with each partition of S. To avoid duplicate joins the join algorithm must be modi-

Algorithm 8.1 Hash-based partitioning

input: Sets R and S, number of partitions k, hash function h
output: Set of pairs fulfilling the join conditions
for all r in R **do**
    $v \leftarrow h(r)$
    $R_v \leftarrow r$
end for
for all s in S **do**
    $v \leftarrow h(s)$
    $S_v \leftarrow s$
end for
for all *partition* R_i of R **do**
    for all *partition* S_i of S **do**
        $output \leftarrow R_i \bowtie S_i$
    end for
end for

fied to only output a join if one of the intervals is not a replica. Pseudocode for domain based partitioning is given in algorithm 8.2.

8.2 Parallel spatio-interval plane-sweep

In [13] a method for partitioning the spatial plane-sweep for parallel processing is described. By using the same method it is possible to parallelize the spatio-interval plane-sweep as well. The goal of the parallel algorithm is to split the domain into several partitions, and then simultaneously process each partition using several parallel sweep lines. The splitting is done by creating k separate sweep lines, where k is the number of partitions. Then for every object r the algorithm checks which sweep line the left and right side of the object corresponds to. If they both correspond to the same sweep line the object is added to the partition belonging to that sweep line. If not, the object is added to the partition belonging to the sweep line that corresponds to the left side of the rectangle. Then for all the sweep lines the rectangle intersects it is copied to the partitions corresponding to those sweep lines as well. Similarly to the technique used in the domain-based partitioning described in the previous section each replica have a flag indicating that it is a replica. Then when the algorithm outputs a join it only includes it if at least one of the two elements is not a replica.

Algorithm 8.2 Domain-based partitioning

input: Sets R and S, number of partitions k
output: Set of pairs fulfilling the join conditions
Split domain of R and S into k tiles
for all r in R **do**
    $t_{start} \leftarrow$ domain tile covering $r.start$
    $t_{end} \leftarrow$ domain tile covering $r.end$
    add r to R_{start}
    for all t_j in $\langle t_{start+1}, t_{end}]$ **do**
        replicate r to R_j
    end for
end for
for all s in S **do**
    $t_{start} \leftarrow$ domain tile covering $s.start$
    $t_{end} \leftarrow$ domain tile covering $s.end$
    add s to S_{start}
    for all t_j in $\langle t_{start+1}, t_{end}]$ **do**
        replicate s to S_j
    end for
end for
for all *partition* R_i of R **do**
    for all *partition* S_i of S **do**
        $output \leftarrow R_i \bowtie S_i$
    end for
end for

Chapter 9

Implementation

The algorithms discussed in chapter 7 and 8 have all been implemented from scratch. The code for these implementations were written in the C++11 programming language. Four different algorithms were implemented. The sequential spatial EBI algorithm, the sequential spatio-interval plane-sweep algorithm and the two corresponding parallel algorithms. The parallel algorithms were implemented using the Pthreads library, which is described more in section 9.2. The parallel spatial EBI algorithm was implemented using the hash-based partitioning described in section 8.1 while the parallel spatio-interval plane-sweep was implemented as described in section 8.2.

9.1 Data structures

The active set in the SEBI algorithm uses the *std::map* class from the C++ standard library. Getting elements and erasing elements are done in time complexity $O(log(n))$, where n is the number of elements in the structure. Adding new elements to the map is done in $O(n)$ time. The *sweep_structure* in the

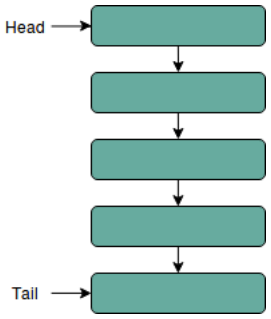

Figure 9.1: Simple linked list

spatio-interval plane-sweep algorithm is implemented using a *simple linked list*. A simple linked list consists of a series of nodes, each containing a pointer to the next node as well as the object itself. Additionally the linked list stores pointers to the first and the last element in the list, the head and the tail. A visualization is shown in Figure 9.1. Insertion into the list is done by creating a new node containing the new object. The current tail is then adjusted to point to the new node and the tail pointer is also adjusted to point to the new node. Removal of an object is done by scanning through the list. When the node to be removed is encountered the previous node is adjusted so that it points to the node after the one to be removed. Inserting into the list is done in $O(1)$ time, while deleting from the list is done in $O(n)$ time.

9.2 POSIX Threads

POSIX Threads, or Pthreads, is a shared-memory parallel programming framework. Shared-memory means that all threads have access to the same memory. This leads to a natural way to reason about the parallel program, but it can also lead to problems with synchronisation and if the programmer is not careful it can lead to unpredictable behaviour and race conditions [14]. Pthreads implements the POSIX standard, which is a specification for a variety of software on Unix-like operating systems. It is implemented as a library that can be linked with C and C++ programs. Threads are initiated by a call to *pthread_create*. One of the parameters supplied to *pthread_create* specifies the function that each thread will execute in parallel. During execution all the threads have access to variables that are declared globally in the program. To avoid race conditions when updating shared structures the Pthreads library also supports the use of mutexes, which are locks set on shared data so that only one thread can update it at the same time.

Chapter 10

Experimental results

In this chapter some experimental results are presented. The tests were executed on a machine running Ubuntu 16.04. The machine had 128 GB of internal memory and an Intel Xeon E5-2640 V3 processor with 8 cores and 16 threads. Most of the tests were done using a data set from the social media platform Twitter. Each row in the dataset contains a single tweet with metadata. The metadata contains a bounding box for the area where the tweet originated as well as a time interval from the tweet was first posted until its first retweet, that is the time from posting until another user has posted a reference to the original tweet. These two properties were used as the spatial and the interval component of each object. To test the spatio-interval join subsets of the whole dataset which were of different sizes where randomly selected and a self-join was executed on the subsets. To reduce the amount of random differences in execution time introduced by the operating system and other processes running on the machine, each test was run ten times and the average of these runs were reported.

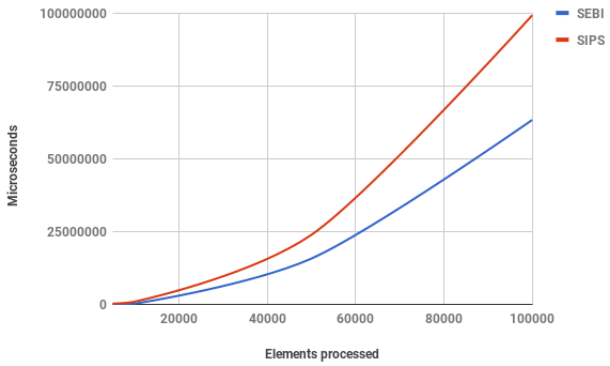

Figure 10.1: Performance of spatial EBI and spatio-intervals plane-sweep

Figure 10.1 shows the performance of two sequential algorithms when increasing

the number of elements present in the join operation. As can be seen from the graph the spatial EBI algorithm outperforms the spatio-interval plane-sweep, and the distance in performance increases as the number of elements increases. This can be a result of the sweep structure implementation. It is implemented using a simple linked list, meaning that the entire sweep structure must be iterated twice in each iteration of the algorithm. The sweep structure is iterated both when the *SEARCH* method and the *REMOVE_INACTIVE* method is called. In the spatial EBI algorithm only one of the active sets needs to be iterated once during each iteration.

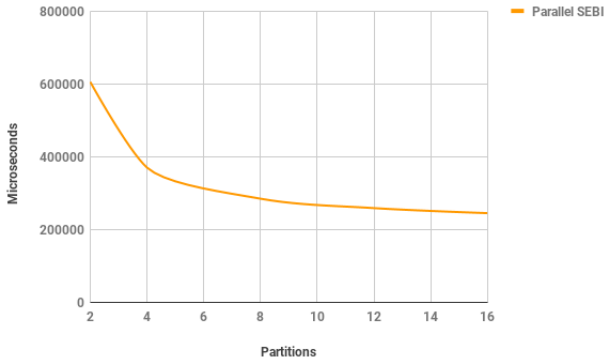

Figure 10.2: Speedup of parallel spatial EBI

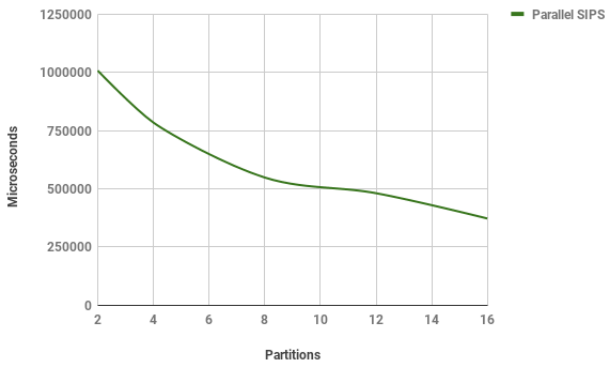

Figure 10.3: Speedup of parallel spatio-interval plane-sweep

The two figures 10.2 and 10.3 show the speedup of the two parallel algorithms when increasing the number of partitions that the input data is split into. The speedup result was generated by a self-join of a subset of 50000 elements from the Twitter data set and varying the number of partitions. Since an individual thread is created for each partition this is equivalent to increasing the number of threads in the processor used. The results show that both parallel algorithms have better performance with more cores, and that the performance gain increases with the number of cores. Figure 10.4 shows a comparison of the performance of the two parallel algorithms run with 16 partitions. The parallel

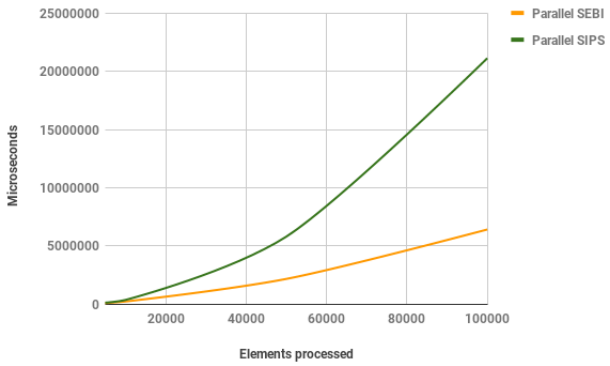

Figure 10.4: Performance of parallel SEBI and parallel SIPS

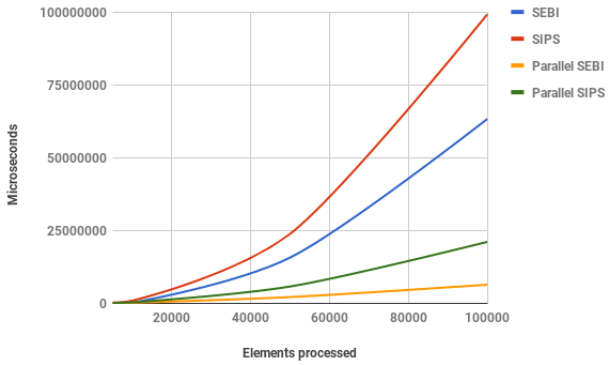

Figure 10.5: Performance of all four algorithms

spatial EBI algorithm outperforms the parallel spatio-interval plane-sweep algorithm quite significantly as the number of elements increases. This is somewhat consistent with the performance of the sequential algorithms as the underlying sequential algorithms are run on every partition in its parallel counterpart. In Figure 10.5 the performance of all four algorithms implemented is compared. Both parallel algorithms perform much better then their corresponding sequential algorithm, and the parallel spatial EBI algorithm is the fastest of the four.

Chapter 11

Future work

The experimental results show that is possible to solve the spatio-interval join by modifying the existing algorithms for spatial join and interval join. It is also possible to significantly reduce the execution time by parallelizing these algorithms using the same parallelization techniques used for the spatial join and interval join algorithms. Of the algorithms presented the parallel spatial EBI algorithm is the most efficient, which is consistent with the performance of the sequential algorithms. The spatial EBI algorithm is not only more efficient than the spatio-interval join, it is also easier to implement as the sweep structure required in spatio-interval plane-sweep requires more implementation then the data structure used for the spatial EBI. The spatio-interval plane-sweep could be made more efficient by implementing the sweep structure using a different data structure as discussed in section 6.4, although this would make it even more complicated to implement.

For future work both the algorithms can be improved by implementing the different optimizations discussed in chapter 6. The spatial EBI algorithm can be made more efficient by implementing a gapless hash-map and by implementing the lazy evaluation described in section 6.1 and 6.2. The spatio-interval plane-sweep can be improved by implementing a more efficient sweep structure. It can also make use of a linear ordering to sort the objects in an order that ensures that objects close to each other are close in the ordering. The algorithms presented here should also be measured against existing spatio-interval techniques like the spatio-interval join from [18] to see how they compare.

References

[1] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, May 1990.

[2] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[3] Panagiotis Bouros and Nikos Mamoulis. A forward scan based plane sweep algorithm for parallel interval joins. *Proc. VLDB Endow.*, 10(11):1346–1357, August 2017.

[4] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using r-trees. *SIGMOD Rec.*, 22(2):237–246, June 1993.

[5] Bhupesh Chawda, Himanshu Gupta, Sumit Negi, Tanveer A Faruquie, L Venkata Subramaniam, and Mukesh K Mohania. Processing interval joins on map-reduce.

[6] Anton Dignös, Michael H. Böhlen, and Johann Gamper. Overlap interval partition join. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1459–1470, New York, NY, USA, 2014. ACM.

[7] Ahmed Eldawy, Louai Alarabi, and Mohamed F. Mokbel. Spatial partitioning techniques in spatialhadoop. *Proc. VLDB Endow.*, 8(12):1602–1605, August 2015.

[8] Ahmed Eldawy and Mohamed F. Mokbel. A demonstration of spatialhadoop: An efficient mapreduce framework for spatial data. *Proc. VLDB Endow.*, 6(12):1230–1233, August 2013.

[9] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984.

[10] Edwin H. Jacox and Hanan Samet. Spatial join techniques. *ACM Trans. Database Syst.*, 32(1), March 2007.

[11] Krishna Kulkarni and Jan-Eike Michels. Temporal features in sql:2011. *SIGMOD Rec.*, 41(3):34–43, October 2012.

[12] Per-Åke Larson and Justin Levandoski. Modern main-memory database systems. *Proc. VLDB Endow.*, 9(13):1609–1610, September 2016.

[13] Mark McKenney, Roger Frye, Mathew Dellamano, Kevin Anderson, and Jeremy Harris. Multi-core parallelism for plane sweep algorithms as a foundation for gis operations. *GeoInformatica*, 21(1):151–174, Jan 2017.

[14] Peter S Pacheco. An introduction to parallel programming, 2011.

[15] Danila Piatov, Sven Helmer, and Anton Dignös. An interval join optimized for modern hardware. pages 1098–1109, 2016.

[16] J. Von Neumann. First draft of a report on the edvac. *Annals of the History of Computing, IEEE*, 15(4):27–75, 1993.

[17] Geraldo Zimbrao, J Moreira de Souza, and V Teixeira de Almeida. The temporal r-tree, 1999.

[18] G. Zimbro, J.M. De Souza, and V.T. De Almeida. Efficient processing of spatiotemporal joins. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2973:190–195, 2004.