University of Pisa and Scuola Superiore Sant'Anna
Master Degree in Computer Science and Networking

Master thesis

Skyline on sliding window data-stream: a parallel approach

Candidate
Salvatore Di Girolamo

Supervisor
Marco Vanneschi

Examiner
Antonio Cisternino

Academic year 2012/2013

Contents

Chapter 1

Introduction

Suppose you are a stock market buyer and your choices, in terms of what blocks of shares buy, are driven by a set of block-relative attributes such as the number of shares in a block and its price. In particular, you are interested in the blocks with the highest number of shares and lowest price. Unfortunately, the two goals are complementary as the blocks with an high volume of shares tend have an higher price: since it is not possible to establish a total ordering among the blocks, it is impossible to choose a set of best solutions. What can be done, instead, is to present a set of interesting ones, leaving to you the choice of what buy. A block of shares is interesting if does not exist another block with higher volume and lower price. The set of the interesting blocks is the *skyline* [21], also referred as *Pareto frontier* [4] or *maximum vector* [10].

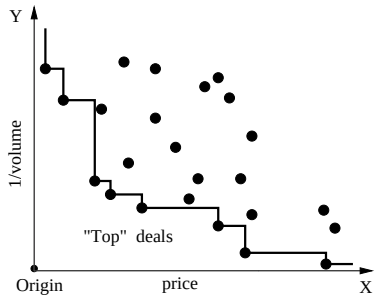


Figure 1.1: Skyline over a set of block of shares [25]

Problem definition In general, given a set of d-dimensional points, the skyline can be computed as the set of points which are not dominated by any other points. A point p is dominated by a point r if this last is not

worse than p in every single dimension, but it is better than p in at least one. The meaning of "better" and "worse" depends on the type of the values represented on a specific coordinate, e.g. in the stock market example, a block of shares is "better" on the x-axis than another one if it has a lower price.

The skyline query has received considerable attention in the database research community since its introduction, due to its importance in applications like decision making and multiple criteria optimization. Moreover the skyline computation problem is also related to a number of well-known problems, such as *convex hulls* [9], *top-K* queries [24], and *nearest neighbor search* [19].

Skyline on data stream Recently, with the arrival of the big data era and the increasing complexity of the available data, we have witnessed to a paradigm shift to query processing over continuous streams. The skyline operator is no more applied to static datasets stored in disks but, instead, the points over which it must be applied are received through a stream with variable arrival rates. Since the stream is possibly unbounded, the skyline cannot be computed over all the points: the approach is to compute the skyline of the most recent points, using sliding window specifications (time-based or count-based). In this case, the output of the module is itself a stream: it is the sequence of the skyline updates. A skyline update indicates the entering or the exiting of a point from the skyline.

As an example of the application of the skyline operator over a stream of d-dimensional points, we can consider again the stock market application: the information regarding blocks of shares are continually updated due to modification of their price and/or volume. Clearly, the stock market buyers are interested only in the most recent top deals and not in the ones raised, for instance, many hours/days ago. The skyline on sliding window data-stream matches perfectly this problem: only the most recent deals, which are not dominated by any other one in the same sliding window, are presented to the user. Another application could be the analysis of the data received from a wireless sensor network. The records are received over a stream and the records which are older than a certain threshold (the sliding window length) must be discarded from the system because they most likely do not reflect the current sensor readings.

Related works In literature there is a number of works regarding skyline queries. These works can be classified in according to two factors:

- if they are centralized or distributed;

- if they operates on static datasets or on streams.

All the four combinations are possible. Centralized skyline computation are proposed both on static datasets [21], [12], [11], [7], [6], and streams [26], [15], [25], [28]. At the best of our knowledge, all the parallel skyline algorithm proposed in literature are relative to only static datasets [2], [13], [22], [3], [23], [20].

Our contribution In this thesis, the Data Stream Processing (*DaSP*) methodologies are applied in order to design a parallel module which applies the skyline operator over a stream of d-dimensional point.

Our work starts considering the two skyline sequential algorithms, operating on stream, described in [26] by Papadias & Tao, namely *Lazy* and *SIT*. The two approaches are compared both from the view point of service time and memory occupancy. The one performing better over all the aspects, SIT, is chosen as the starting point of our parallelization. Since the sequential algorithm presents an high variable load, its parallelization is characterized by the load unbalancing problem. We deal with this problem proposing partitioning techniques aiming to reduce it to negligible levels. Moreover, we investigate on the underlying data structure used in the sequential version, the R-Tree, in order to analyze its efficiency with respect to the employed parallelization pattern, proposing an alternative approach for storing the data. Since the main target is to design a parallel module with high throughput and low latency, an optimization referred as *asynchronous reduce* is introduced. At the end of Chapter 4, we made a theoretical discussion about the maximum sustainable bandwidth of our parallelization or, in other words, the stream arrival rate that the parallel module is able to sustain before it becomes bottleneck, fixed a parallelism degree. As it will be shown in Chapter 5, we reached interesting results in terms of service time and scalability: with the adoption of the discussed load-aware partitioning schemes, the proposed parallel module presents a scalability near to ideal one. The experiments are performed on a commodity Intel multicore made available by the University of Pisa, the detailed description of this architecture is reported in Chapter 5.

The sliding window specification adopted by SIT, and hence by the proposed parallelization, is the time-based one. It is worth noting that, as discussed in [26], this solution is more general then the count-based one, since it can emulate this last using a simple transformation.

Paper organization The rest of this paper is organized as follows. Chapter 2 gives a formal definition of the problem of computing the skyline over

a stream, exposing what are the properties of this type of computation. In Chapter 3 the *Lazy* and *SIT* algorithms are discussed and compared. In Chapter 4 our parallelization of the *SIT* algorithm is presented. In Chapter 5 we experimentally evaluates the performances of the proposed parallelization. Chapter 6 concludes this work with directions for future works.

Chapter 2

Problem definition

2.1 Preliminaries

In this chapter we discuss the skyline problem and the associated properties. Given a d-dimensional data space S, a point r is represented as $r = \{r_1, ..., r_d\}$ where r_i is the value of r in the i-th dimension.

Assuming that there is a total order relationship on each dimension, in this paper we use '$<$' relation, we can define the point dominance relationship as:

Definition 2.1 (Point dominance)**.** Given two d-dimensional points $x = (x_1, x_2, ..., x_d)$ and $y = (y_1, y_2, ..., y_d)$, x dominates y (indicated with $x \prec y$) *iff* $\forall i \in [1, d]\ x_i \leq y_i$ and $\exists j \in [1, d] \mid x_j < y_j$.

From now on, if x dominates y we refer to x as a "dominator" of y. For each point r in the data space we can identify two regions: the *dominance region* ($r.DR$) and the *anti-dominance region* ($r.ADR$). Any point falling in $r.DR$ is dominated by r while any point falling in $r.ADR$ dominates r .

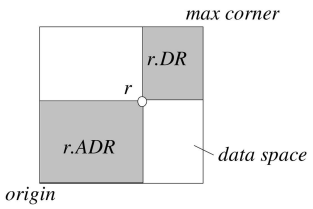


Figure 2.1: 2D dominance and anti-dominance region of r

In Figure 2.1 is reported a representation of this two regions in the 2D case: $r.DR$ is an axis-parallel rectangle whose major diagonal is decided by

r and the "max-corner" of the data space. The anti-dominance region is still an axis-parallel rectangle but its major diagonal is decided by r and the origin of the data space. If a point does not fall neither in $r.DR$ nor in $r.ADR$ then it is said "incomparable" with r.

Definition 2.2. (Skyline) Given a d-dimensional data space S and a set of points P in S, the skyline of P, indicated with $sky(P)$, is the set of points belonging to P which are not dominated by any other point in P. In other words, it is the set of points whose *anti-dominance region* is empty.

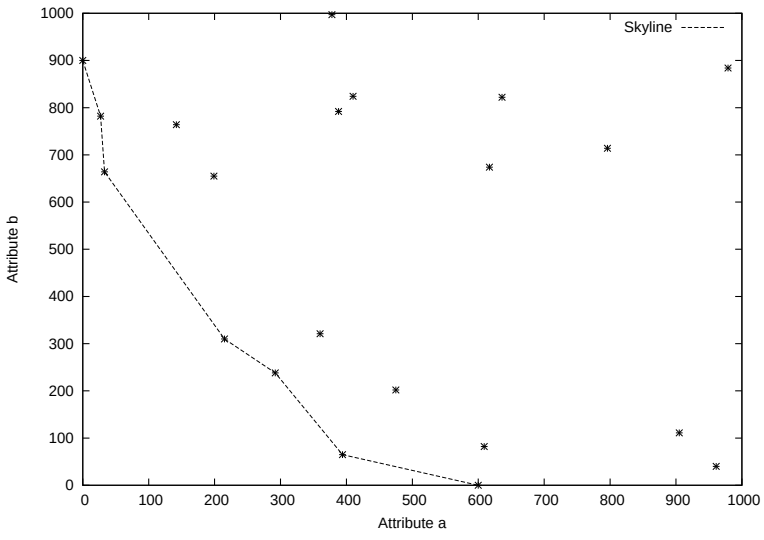


Figure 2.2: Example of a 2-d skyline

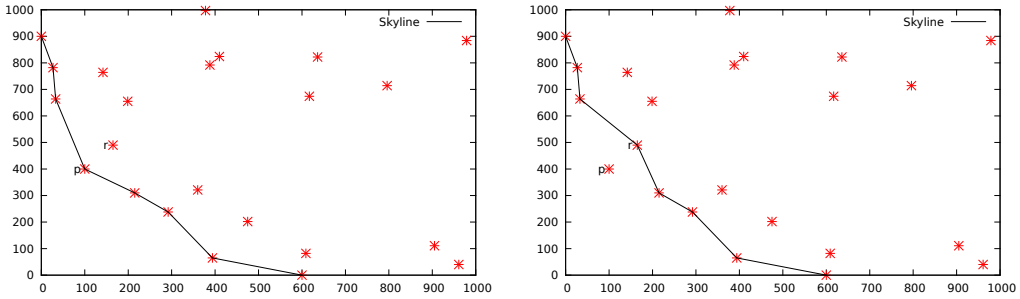
In Figure 2.2 an example of two-dimensional skyline is shown: does not exists a point which is better than r on one or more dimensions without leading to be worse in at least one. This implies that there cannot exist non-skyline points which are better, in every dimension, than skyline ones.

2.2 Skyline over sliding window

Consider the case in which the skyline must be computed over a stream, possible unbounded, of d-dimensional points. Since the number of received points could be infinite, it is impossible to calculate the skyline over all

the entire stream. The idea is to provide the most recent information, computing the skyline over the points received in the last W seconds, where W is a system parameter representing the sliding window length. Fixing a time instance t, the skyline at time t is computed over all the points whose arrival times are in the interval $[t - W, t)$. As the time passes, the window slides causing the expiration of older points. In this model, a point p arrives to the system at $p.t_{arr}$ and expires at $p.t_{exp} = p.t_{arr} + T_w$.

Consider the hypothetical case in which we want to calculate the skyline over the entire stream: once a point p is designed as non-skyline point, it can be immediately discarded because, since it is not a skyline point, there exists at least one skyline point which dominates it. In the sliding window model things are different: all the points expire either they are into the skyline or not. As a consequence, when all the points dominating p will be expired, it will become a skyline point. This behavior implies that, differently from the previous case, when p arrives, we cannot discard it simply because it is not a skyline point at that time, since it could enter into the skyline in the future.



(a) At time t with $p.t_{arr} < t < p.t_{exp}$ (b) At time t' with $t' \geq p.t_{exp}$

Figure 2.3: Effects of a point expiration over the skyline

In Figure 2.3 is reported an example of a point expiration: at time t the skyline is the one in Figure 2.3a, the only point which dominates r is p. At time t', p is expired (Figure 2.3b) and, since there are no more points dominating r, this last becomes a skyline point. In this example we assume that no other points expire in $[t, t']$.

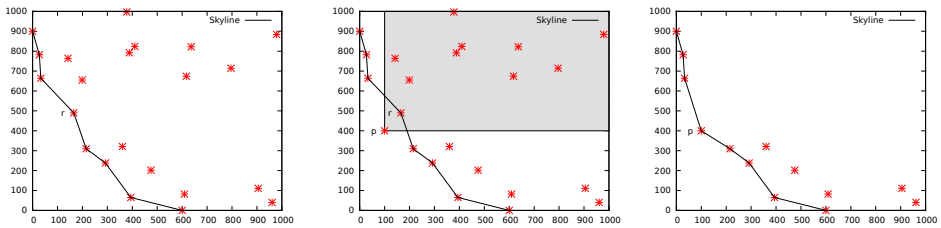
From what described above we can extract the following property:

Property 2.1. An arriving point p cannot be discarded from the system even if it is not a skyline point at its arrival time. It will become a skyline point when all its dominators will be expired.

An arriving point, either it is a skyline point or not, describes a set of points (possibly empty) which are dominated by it: is the set of points falling in $p.DR$, where p is the arriving point. The points belonging to this set can be safely discarded from the system when p arrives since they have no more chances to enter into the skyline. For example, consider a sliding window containing two points, p and r, with $p.t_{arr} > r.t_{arr}$ and p dominating r. When p arrives to the system there are two possible cases:

- r is a skyline point. Since p dominates r, this last is removed from the skyline and p becomes a new skyline point;
- r is not a skyline point. There are no direct implications on the skyline. The point p will enter into the skyline only if it is not dominated by any other point.

In both cases, the point r can be discarded from the system. By definition, r will enter into the skyline when all its dominators will be expired but, since p (which is a dominator of r) is younger than r, it will expire after r: r will become a skyline a point after its expiration, which is impossible.



(a) $t < p.t_{arr}$ (b) $t' = p.t_{arr}$ (c) $t'' \geq p.t_{arr}$

Figure 2.4: Effects of point arrival

In Figure 2.4 an example of the effects caused by a new point arrival is shown. In this example it is supposed that there are no expiration in the time interval $[t, t'']$. At time t (2.4a) the point p is not arrived yet. In 2.4b, the dominance area of p ($p.DR$) is showed in gray: the points falling in that region can be discarded (pruned) from the system either they are skyline points or not. In Figure 2.4c the new configuration is showed: p is now part of the skyline and all the points dominated by it are discarded. Please note that even if the incoming point is not a skyline one (such as in this case), it is always possible to discard the points falling in its dominance region.

From what said above, we can extract the following properties:

Property 2.2 (Pruning)**.** Once a new point p arrives, all the older points dominated by it (i.e. falling in $p.DR$) can be safely discarded from the system since they have no more possibilities to become skyline points. We refer to the points that can be pruned as "obsolete" points.

Property 2.3 (Skyline inheritance)**.** If a newly arrived point dominates one or more skyline point, then it is a new skyline point.

This last follows from a simple observation: if r is a skyline point, then its anti-dominance area ($r.ADR$) is empty; when a new point p dominating r arrives, its anti-dominance area will be contained in $r.ADR$ and thus empty even it. Since a point belongs to the skyline if its ADR is empty, we can conclude that p will be a skyline point while r will exit from the skyline and, by Property 2.2, will be discarded.

We previously said that a non-skyline point become a skyline one when all its dominators are expired. We can refine this definition using the concept of "critical dominance":

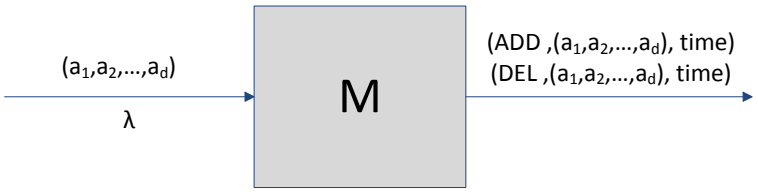
Definition 2.3 (Critical dominance)**.** A dominance relation $x \rightarrow y$ is critical *iff* x is the youngest point (but older than y) dominating y. We use $x \xrightarrow{c} y$ to indicate that "x critically dominates y".

Since the critical dominator of a point p is the youngest among the p's dominators and thus it is the one which expires later, we can state that a point will enter into the skyline when its critical dominator will be expired. It is worth noting that, among the dominators of a point, the critical dominance relationship is not related to the spatial coordinates of these but only to their arrival times.

The critical dominance relationship has not to be confused with the exclusively dominance one. If a point p is exclusively dominated by a point r, it means that r is the unique dominator of p. Instead, the critial dominance has no relation with the cardinality of the dominator's set of p.

2.3 Module definition

In this section we give a first description of a module M which applies the skyline operator over a stream of tuples and produces a stream of skyline updates.



The module M receives a stream of d-dimensional points with a rate of λ tuples/sec. The module applies the skyline operator over the points received from the input stream adopting the sliding window specifications. The temporal length of the sliding window (W) is a system parameter. Since we adopt a temporal sliding window, the number of tuples received in a window is:

$$K = \lambda \times W$$

Each one of these points could be or become a skyline point (by Property 2.1) meaning that M must be able to accommodate K tuples per sliding window.

Property 2.2 states that when a point become obsolete, it can be discarded from the system even if it is not expired yet. This behavior, called "pruning", is optional and it does not have any impact on the correctness of the skyline. Please note that it is independent from the action of updating the skyline when a skyline point become obsolete. This last action is essential for the correctness of the computation and it is different from discard obsolete points from the system (either they are skyline or not), which has consequences only on the memory utilization. If M adopts pruning techniques, then K become an upper-bound to the number of tuples that M must be able to accommodate. This upper-bound will be reached in the unlucky case where, in a window, for each two consecutive points arrival, the older point always dominates the younger one, and hence no pruning is possible.

The module M outputs skyline updates that are expressed in the format:

$$(action, point, timestamp)$$

where the *action* indicates if the *point* enters (+) or exits (−) from the skyline at the specified *time*.

Having described the input/output format specifications, we can define when M has to produce skyline updates:

- If a new point p arrives at time t and, at that time, it is not dominated by any other points in the system, an update $(+, p, t)$ must be produced.
- If at certain time t a point p, with $p.t_{exp} > t$, has no more dominators (since they are all expired), then an update $(+, p, t)$ must be produced.
- If p is a skyline point and the time $p.t_{exp}$ is reached, then an update $(-, p, p.t_{exp})$ must be produced.
- If p is a skyline point and at time t a new point r arrives with $r \prec p$, then p become obsolete and an update $(-, p, t)$ must be produced.

A point arrival can lead to the output of zero, one or more updates. If the incoming point is not a skyline point, then zero updates will be produced. In the other case, the arrival of the new point will lead to the production of $1 + c$ updates, where c is the number of skyline points pruned by the new one. The module can also produce outputs independently from reception events; this occurs in two cases: if a skyline point expires and if a point enters into the skyline since all its dominators are expired.

Supposing that the module is not a bottleneck, its output rate depends on three factors: the input rate, the spatial distribution of the received points and the sliding window length. The first is obvious, the second impacts on the term c which is the mean number of skyline points pruned per received point, while the third determines the points' expire time and hence how often an update for the expiration of a skyline point is emitted.

Output chronological order There are two scenarios which must be considered:

1. the updates are used in a real-time fashion. In this case the skyline updates must be emitted respecting the chronological order: if an update is emitted on the output stream before another one, then the same order must be respected also on their respective timestamps.
2. the updates are used off-line: for example they are stored for future uses. In this case we can charge the consumer itself for the temporal reordering of the skyline updates. This case is valid only if the stream is bounded.

The consumer builds the skyline over the updates sent by M. If we are in the first case, in which the updates are used in real-time, and the consumer receive non ordered updates, it could consider as skyline a set of point that is not a skyline by definition. For example, consider two point a and b with b dominating a and $b.t_{arr} > a.t_{arr}$. Clearly, since one dominates the other, these two points cannot be part of the skyline at the same time. Consider the following sequence of events:

1. at time t, a is part of the skyline and the relative update has been emitted, hence also the consumer knows that a is a skyline point.

2. at time $a.t_{exp}$, the update indicating that a is no more a skyline point is emitted.

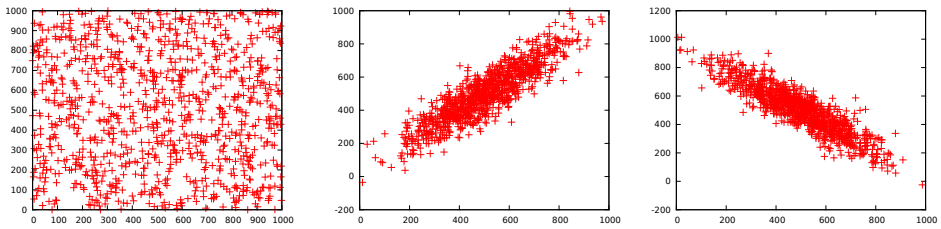
3. at time $b.t_{arr}$, an update indicating that b enters into the skyline is emitted.

The above sequence is correct: a and b are both skyline point but in non overlapping time intervals. Now consider the case in which the updates are not chronologically ordered and, for example, the last two events are inverted: during the time interval between the reception of the two events, the consumer see that a and b are skyline points at the same time, so the skyline that it views is wrong.

From now on, we consider only the case in which the module must guarantee the chronological order of the updates. This behavior is always correct and allows us to handle both the cases in which the updates are used in real-time or not.

2.4 Data distributions

In the skyline problem the spatial distribution followed by the received points plays an important role. In [21] a set of three critical types of data distributions, stressing the effectiveness of the skyline methods, is proposed: *independent*, *correlated* and *anti-correlated*.



(a) Independent (b) Correlated (c) Anti-correlated

Figure 2.5: Data distributions in a two-dimensional space

Correlated distribution It is the case with the lowest number of skyline points: there is a small number of points dominating all the others. If pruning techniques are adopted, this distribution leads to a very small number of stored points, since it is most likely that an arriving point causes the pruning of a large number of already stored ones.

Anticorrelated distribution It is the opposite of the correlated case. A large number of points is into the skyline: each skyline point dominates only a small portion of the entire dataset. It is considered the most challenging of the three since it leads to store a high number of points, even if pruning techniques are adopted.

Independent distribution This is the most general case: the points do not follow a particular data distribution but are randomly distributed over the data space.

Chapter 3

Sequential algorithms

Any implementation of the module described in the previous chapter must implement the following functions:

1. *Data processing*. It is executed at each point arrival: its basic task is to verify if the incoming point belongs to the skyline and, in case, emit a proper update and discard all the skyline points dominated by the new one.

2. *Data maintenance*. The aim of this function is to keep the skyline updated. It ensure that the sliding window specification are respected: when a skyline point expires, the proper update indicating its exiting from the skyline must be emitted. In addition, depending on the specific algorithm, it is also in charge to determine the set of points (eventually empty) to promote to skyline points as a consequence of the expiring of such a point.

The two functions are associated to the two types of activation of the module: the first corresponds to the external activation while the second to the internal one.

In [26], the proposed sequential algorithms are presented as subdivided in two classes: lazy and eager. The lazy strategy delays most computational work until the expiration of a skyline point. The eager method, instead, takes advantage of precomputation to minimize the memory consumption. The same distinction is adopted in the following discussion.

3.1 Lazy algorithm

This algorithm is presented in [26]. It is based on the idea that each function, namely *data processing* and *data maintenance*, must performs only the

strictly necessary operations to full-fill their tasks: the former must only verify if the incoming point is a skyline point while the latter is in charge of discard the expiring point and determine the set of points to promote to skyline point as a consequence of its expiring.

The authors introduce two data structures, DB_{sky} and DB_{rest}: in the former are stored points that currently are part of the skyline, while in the latter are stored points that do not belong to the current skyline. If a point p is stored in DB_{rest}, it means that p is dominated by one or more points of DB_{sky} and by zero, one or more points of DB_{rest} When all the dominators of p expire, p must be moved to DB_{sky} and thus promoted to skyline point. The two DB are represented using R-Trees [1].

For the sake of completeness we cite a similar algorithm presented in [15]. It differs from Lazy in the employed data structures: the concepts of DB_{sky} and DB_{rest} are merged, storing all points in an unique data structure. Finally the authors present and compare the implementation of their algorithm with two different types of data structures: R-Trees and Quadtrees.

3.1.1 Data processing

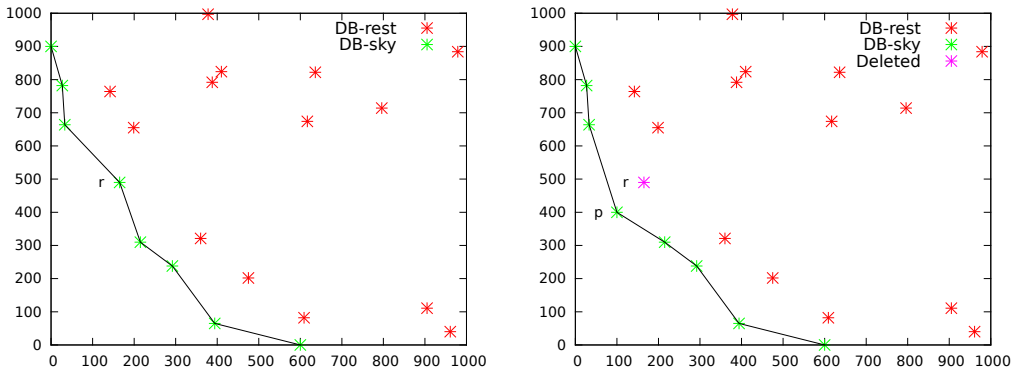
When a point arrives, the *data processing* function must be executed: the first step is to check if the incoming point p is dominated by someone other. If yes, p is simply inserted in DB_{rest} and no other operations are performed. Otherwise, if there are no points dominating p ($p.ADR$ is empty), p is a skyline point, hence it is added to DB_{sky}. In this last case, the algorithm checks if p dominates some skyline points, which have to be removed from the skyline and discarded from the system.

Algorithm 1 Lazy Data processing. Executed when a new point p arrives.

```
1: if isDominated(p, DB_sky) then
2:    insert p in DB_rest
3: else
4:    for each r ∈ findDominatedPoints(p, DB_sky) do
5:       delete r
6:       emit (−, r, current_time)
7:    end for
8:    insert p in DB_sky
9:    emit (+, p, current_time)
10: end if
```

The Algorithm 1 shows the pseudocode of the *data processing* function. At line 1, the function $isDominated()$ checks if the new point is dominated

by anyone in DB_{sky}. The function $getDominatedPoints(p, DB)$ returns all the points in DB dominated by p. The pseudocode of this last is reported in Appendix B.



(a) $t < p.t_{arr}$ (b) $t' = p.t_{arr}$

Figure 3.1: Effects of point arrival

For example, in Fig. 3.1, when p arrives (in 3.1b) it is not dominated by any point, thus the function $isDominated(p, DB_{sky})$ will return false: p must be inserted into the skyline and the point r, which is a skyline point dominated by p, must be deleted.

Please note that the Algorithm 1 does not executes any pruning operations on DB_{rest}. As shown in Fig. 3.1b, the points in DB_{rest} are not pruned even if they became obsolete due to the arrival of p. They will be discarded from the system only when they will reach their expiration time.

3.1.2 Data maintenance

The *data maintenance* function is executed when the oldest skyline point expires, let us call this point p and its expiration time $p.t_{exp}$.

The first step is to remove p from the skyline with the emission of a proper update on the output stream. Moreover the DB_{rest} is cleaned-up, removing all the expired points (which expiration time is less than $p.t_{exp}$).

The next step is to verify if there are points in DB_{rest} that must be promoted to skyline points as consequence of the expiring of p. These points, if they exist, are the points exclusively dominated by p. For finding them, the algorithm performs the following operations:

1. find the set of points in DB_{rest} dominated by p, call it D_p;

2. find the sub-skyline over the points in D_p, let us call it the *miniSkyline*;

3. verify for each point c in *miniSkyline* if it belongs to the skyline or not. If yes, a proper update is emitted indicating the entrance of c into the skyline at time $p.t_{exp}$.

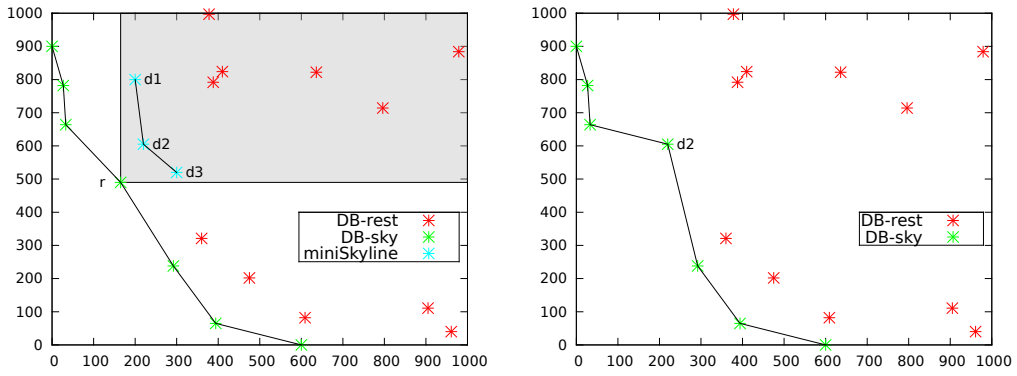
Note that the step 2 is optional, through it we try to reduce the number of points which have to be analyzed in step 3. As an example, if x and y are two points in D_p and $x \prec y$, we already know that y does not belong to the skyline, since its ADR is not empty. If step 2 is performed, y will not be checked in step 3 because it will not belong to the *miniSkyline*.

In [15] is presented an algorithm capable to merge the first two steps, it is called *MINI*. Given a point c and a set of point P represented though an R-Tree (in our case DB_{rest}), *MINI* finds the skyline over the points in P with the constraint that only the ones dominated by c must be taken in account, in other words it finds the skyline over the points in P falling $c.DR$. The pseudocode of this algorithm is reported in Appendix A.

Algorithm 2 Lazy data maintenance function. Executed when the oldest skyline point p expires (at time $p.t_{exp}$).

```
1: remove p from DB_sky
2: emit (−, p, p.t_exp)
3: for each s ∈ DB_rest | s.t_exp ≤ p.t_exp do
4:    delete s
5: end for
6: compute the miniskyline for the set of data in DB_rest dominated by p
7: for each s ∈ miniskyline do
8:    if not isDominated(s, DB_sky) then
9:       move s from DB_rest to DB_sky
10:      emit (+, s, p.t_exp)
11:   end if
12: end for
```

Algorithm 2 reports the pseudocode of the data maintenance function executed by Lazy. In Fig. 3.2 an example of how this function works is reported. When the skyline point r expires, the *miniSkyline* is calculated over all the points in DB_{rest}. In Fig. 3.2a, the *miniSkyline* is composed by the points $\{d1, d2, d3\}$. Please note that not all the points belonging to this set will become skyline ones: only the ones that are not dominated by any other skyline points will become (in the example only the point $d2$). The Fig. 3.2b shows the configuration after the expire of r.



(a) $t = r.t_{exp}$ (b) $t > r.t_{exp}$

Figure 3.2: Effects of point expiration

3.2 Eager algorithms

The idea on which the algorithms of this class are based is to introduce an additional processing phase executed at each point arrival, in order to reduce the time needed by the maintenance phase. Moreover, eager is more "aggressive" from the point of view of the memory occupancy: it tries to free memory as soon as possible, adopting pruning techniques to expunge obsolete points from the system.

3.2.1 Skyline influence time based: SIT

This algorithm is described in [26] and it is based on the concept of point's skyline influence time. For correctness purposes, SIT needs to remove obsolete points, either they are skyline ones or not, at each point's arrival.

Definition 3.1. The skyline influence time of a point p, indicated with SIT_p, is the time instance indicating the entering of p into the skyline. It is defined as the expiring time of the point which critically dominates p (see def. 2.3): if $c \xrightarrow{c} p$ then $SIT_p = c.t_{exp}$.

When a point arrives, it is possible to establish which is its critical dominator and thus the value of SIT_p. If the critical dominator does not exists, SIT_p is equals to 0 indicating that p becomes immediately a skyline point. Subsequent arrivals cannot influence SIT_p, except the case in which the incoming point dominates p: in this case p cannot more enter into the skyline so its SIT become $+\infty$ and it must be discarded from the system. Therefore, once the skyline influence time of a point p is determined, only two situations are possible: the time SIT_p is reached and then p must be inserted into the skyline or a new point c, which dominates p, arrives and then p must be discarded. This behavior allows us to define an important property, which is valid only if the pruning of the obsolete points is performed at each arrival:

Lemma 3.1. *If a point p reaches its expiration time then, at that time, it is part of the skyline.*

Proof. If not, it means that a point c, which dominates p, exists and its expiration time is after the one of p. But, since expiration times are defined as $c.t_{exp} = c.t_{arr} + W$ and $p.t_{exp} = p.t_{arr} + W$, it also means that c is arrived after p ($c.t_{arr} > p.t_{arr}$): this is impossible because, in that case, being c younger than p and dominating it, p would have been discarded from the system when c is arrived and so its expire event would not have existed. □

When a point reaches its expiration time is simply removed from the skyline and no other actions are required: the expiration time of p will match with the SIT of the points critically dominated by it, which would enter into the skyline. By the following lemma, we know that these points are exactly the ones that have to be inserted into the skyline as a consequence of p's expiration.

Lemma 3.2. *When a point expires the points which it critically dominates are also the ones exclusively dominated by it.*

Proof. The point p is the critical dominator (the youngest one) of a set of points, let us call this set C_p. Since p is an expiring point, all other dominators of C_p[1] are already expired (they are older than p): p is the unique dominator of the points in C_p not expired yet, in other words the points in C_p are the ones exclusively dominated by p. □

Differently from the lazy approach, now all the points are stored in an unique data structure, let us call it DB, which is still implemented with an R-Tree. This choice is justified by the different behavior of the two algorithms when a new point arrives.

- In SIT, we have to search the new point's critical dominator among all points. In lazy, instead, we have only to check if it is dominated by some skyline points.
- In Lazy, no pruning operations on DB_{rest} are performed but only the point eventually dominated in DB_{sky} are discarded. In eager, we search the points dominated by the incoming one, and thus to be discarded, among all points currently in the system.

The two observations motivate why in lazy it is useful to maintain a distinction between the points currently in the skyline and the ones that do not belong to it, while in SIT this distinction is meaningless.

Event list SIT maintains an event list EL where each event is a triple (p, t, tag) where p is a point, t is the time at which the event must be triggered and tag indicates the type of event. There are two types of events: $skytime$ and $expire$. The event $(p, t, skytime)$ indicates the entering of p into the skyline at time t, thus t is SIT_p. Instead, the event $(p, t, expire)$ indicates the expiring of p at time t, thus t is the expiration time of p ($p.t_{exp}$).

[1] all other dominators of each point in C_p

Algorithm 3 SIT data processing function. Executed when a new point p arrives.

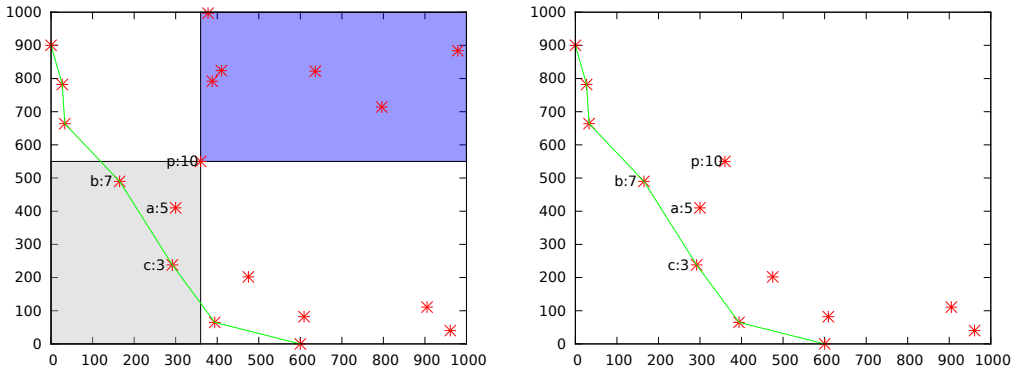
1: $D_p = getDominatedPoints(p, DB)$
2: **for each** c in D_p **do**
3: delete c from DB
4: delete the event associated with c (ev_c) from EL
5: //if $ev_c.tag = expire$ then c is a skyline point
6: **if** $ev_c.tag = expire$ **then** emit $(-, c, current\_time)$
7: **end for**
8: insert p in DB
9: $criticalDominator = findCriticalDominator(p, DB)$
10: **if** $criticalDominator$ is null **then**
11: $//p$ belongs to the skyline immediately
12: emit $(+, p, current\_time)$
13: insert $(p, p.t_{exp}, expire)$ into EL
14: **else**
15: $SIT_p = criticalDominator.t_{exp}$
16: insert $(p, SIT_p, skytime)$ into EL
17: **end if**

Data processing In Algorithm 3, the pseudocode of the SIT data processing function is reported. The steps carried out by this function, when a new point p arrives, are the following ones:

- pruning: delete all points in DB dominated by p and their associated events from EL. Since the event $(c, c.t_{exp}, expire)$ is inserted in EL only when c becomes a skyline point, at line 6 we can check if the pruned point belongs to the skyline or not simply checking if the tag of its associated event (ev_c) is *expire* or not. If a point is a skyline point, its removal must be notified onto the output stream;
- insert p in DB;
- find the point c which critically dominates p. If it exists, insert the event $(p, c.t_{exp}, skytime)$ into EL (recall that, in this case, $SIT_p = c.t_{exp}$). Otherwise, if c does not exists, p is a skyline point: notify this fact onto the output stream and insert the event $(p, p.t_{exp}, expire)$ into EL.

The operation of finding the critical dominator of a point is reported in [26] with the name *d-sided max search*. From now on, we will use the

notation $findCriticalDominator(r, P)$ to indicate such operation. Specifically, it indicates the operation of finding, among all the points in P, the one which critically dominates r, if it exists. Its pseudocode is reported in Appendix C.



(a) $t = p.t_{arr} = 10$ (b) $t > p.t_{arr}$

Figure 3.3: Point arrival example. The label $a : t$ indicates that the point a is arrived at time t.

In Figure 3.3 is reported an example of point arrival. At time 10 the point p arrives (Fig. 3.3a), its dominators are the one in $p.ADR$ (indicated by the gray rectangle). The critical dominator of p is its youngest dominator, in this case the point b: the SIT of p is set to the expire time of b, hence $SIT_p = b.t_{exp} + W = 7 + W$. On the other hand, all the points falling in $p.DR$ (indicated in blue), must be pruned. The results of the processing of p is shown in Fig. 3.3b.

Algorithm 4 SIT data maintenance function. Executed as a trigger for the oldest event e (with time $e.t$) in EV.

1: delete e from EL
2: **if** $e.tag = expire$ **then**
3: emit $(-, e.point, e.t)$
4: delete $e.point$ from DB
5: **else**
6: $//e.tag$ is $skytime$
7: emit $(+, e.point, e.t)$
8: insert $(e.point, e.point.t_{exp}, expire)$ into EL
9: **end if**

Data maintenance The maintenance function of SIT is pretty trivial: it has to monitor the event list executing the events in chronological order. When an event is triggered, there are two possibilities:

- it is an *expire* event: the associated point is discarded from the DB and its removal from the skyline is notified on the output stream (by Lemma 3.1 we know that, since it is an expiring point, it is also a skyline point).
- it is a *skytime* event: the associated point p must be added to the skyline (emitting a proper update on the output stream) and a new event, indicating the expiration of p, must be inserted into EL.

3.2.2 Critical dominance based: DOM

This algorithm is an our proposal. It is based on what discussed in [25] and relies on the same concepts exposed by SIT. The basic idea it to store the non-skyline points directly in a list associated to their critical dominators. When a point expires, all the points critically dominated by it are promoted to skyline points.

As in SIT, an unique data structure DB for storing all points (either skyline points or not) is used. Even in this case, DB can be implemented with an R-tree.

At each point p are associated the following fields:

- a flag *is_skyline* that indicates if it is a skyline point or not;
- a list, called *criticalList*, storing references to the critically dominated points;
- a reference to its critical dominator. If p is a skyline point this field contains a null value.

Data processing When a point p arrives, it is inserted in DB and its critical dominator is searched. If it does not exists then p become immediately a skyline point, otherwise a reference to p is inserted in the *criticalList* of its critical dominator. Moreover, all the points which are become obsolete due to p's arrival are pruned (enabling Lemma 3.1). If some of the pruned points are skyline points, then their removal from the skyline is notified on the output stream. It is worth noting that when a point is pruned it must be also deleted from the *criticalList* of its critical dominators.

The Algorithm 5 shows the pseudocode of the processing module, executed at the arrival of each point. Differently from SIT, in the pruning

Algorithm 5 DOM processing phase. Executed when a new point p arrives.

```
1:  D_p = getDominatedPoints(p, DB)
2:  for each c in D_p do
3:      delete c from DB
4:      if c.is_skyline then emit (−, c, current_time)
5:      else delete ref to c in the criticalList of c.criticalDominator
6:  end for
7:  insert p in DB
8:  cdom = find the point which critically dominates p
9:  if cdom is null then
10:     //p belongs to the skyline immediately
11:     p.is_skyline ← true
12:     emit (+, p, current_time)
13: else
14:     p.criticalDominator = cdom
15:     insert p into cdom.criticalList
16: end if
```

phase (lines 2-6), we do not have any method to distinguish if a point belongs to the skyline or not: this is why a flag, indicating the point status, has been introduced.

As an example of the execution of the data processing function, consider again the Figure 3.3. The newly arrived point p is inserted in the *criticalList* of its critical dominator, which is b. As in SIT, all the points dominated by p are pruned.

Data maintenance The data maintenance function is executed only when a point p expires. When this happens, p is removed from the skyline (at that time p is for sure a skyline point by Lemma 3.1) and all the points in its *criticalList* are promoted to skyline points. By Lemma 3.2, we know that these point are exactly the ones that must enter into the skyline as a consequence of p's expiration.

As said, this algorithm is based on SIT. However, its implementation require the adoption of one data-structure for each stored point and the pruning operation are more costly than SIT, since the reference to the pruned point must be searched in the *criticalList* of its critical dominator. Due to this motivation, this algorithm is not taken in account for the parallelization. It is reported only for the sake of completeness.

Algorithm 6 DOM data maintenance. Executed when a point (p) expires.

1: delete p from DB
2: emit $(-, p, current\_time)$
3: **for each** $c \in p.criticalList$ **do**
4: $\quad c.is\_skyline \leftarrow true$
5: $\quad$ emit $(+, c, current\_time)$
6: **end for**

Chapter 4

ParSIT

The sequential module has a fixed offered bandwidth which it is able to sustain. If the input rate grows, becoming greater than the module offered bandwidth, its utilization factor become greater than one and the module will be a bottleneck. In the case of skyline on stream, the bottleneck does not represent only a performance problem but it impacts also on the produced skyline. In fact, if the utilization factor is greater than one, the sequence of the skyline updates emitted by M is semantically correlated to it. As said in Section 2.3, the module has two type of activations:

- External activation. When a point is received from the input stream, the module is activated for the processing of the new point;
- Internal activation. The module is internally activated in two cases: a) when a point expires; b) when the skyline influence time of a point is reached. This type activation does not depend on the reception of new points from the input stream.

Consider two points, l and p: the former is a point already stored in the system, its skyline influence time is set to t'; the latter is an incoming point with an arrival time equals to t. The point p dominates l and $t < t'$ which means that p will arrive before the skyline influence time of l. If the module is not a bottleneck, then the point l will be pruned due to the arrival of p, which is an younger point dominating it: l will never appear in the skyline. Otherwise, if the module is bottleneck, the point p will be elaborated at time $t + \tau$, where τ is the delay introduced by M. In other words, it is the necessary time to serve all the points arrived before p. If $t + \tau \geq t'$, the point l will enter into the skyline before the arrival of p, and only when p will arrive, at time $t+\tau$, it will be pruned. The skyline updates produced in the two cases are different: in the former l will never enter into the skyline

while in the latter it will be part of the skyline during the time interval $[t', t + \tau)$.

The above discussion motivates the need of a parallelization of the module M, in order to make it able to sustain higher arrival rates respect to the sequential version.

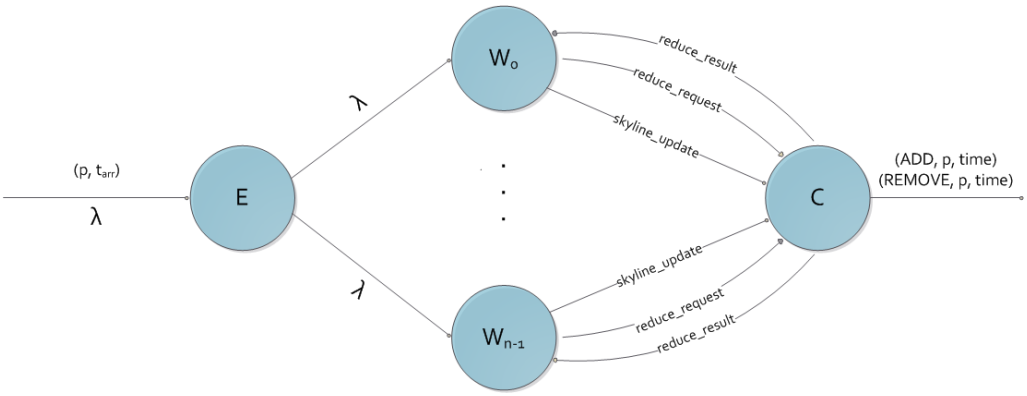


Figure 4.1: ParSIT parallel module definition

The idea is to design a parallelization of one of the sequential algorithms presented in Section 3. The *SIT* algorithm has been chosen since, respect to *Lazy*, it is optimal from the view point of the memory occupancy: it keeps only the points having chances to enter into the skyline, discarding all the "obsolete" ones. In addition, already in the sequential version, *SIT* outperforms *Lazy* in terms of service time.

The employed parallelization pattern is a data-parallel map followed by a reduce phase. As known, a data-parallel map consists in function replication and data partitioning [16]. The functionality of the sequential SIT module are replicated over a set of nodes, the workers. The internal state of the sequential module, composed by all the non-obsolete points in the current sliding windows, is partitioned among these workers. The internal state partitioning is done by selecting, for each incoming point a worker, from now on called the owner, which will be responsible for all the operation concerning it, such as storage, expiration and skyline entrance.

Moreover, to be consistent with the sequential SIT, for each new point, all the points dominated by the new one must be pruned. Since the points are partitioned among the workers, each worker must perform the pruning operation over the points contained in its partition. This leads to an important aspect: each new point must be sent to all the workers, through a multicast communication, but only one of them will be designed as the

owner of it. It is worth noting that the pruning operation is a requisite for the correctness of the SIT algorithm. This behavior is a potential source of load unbalancing among the workers: an arrival of a new point can lead one or more worker to empty, or to drastically reduce, its partition.

If the partitioning schema relies on the spatial distribution of the incoming points, there could be another potential source of load unbalancing, as explained in the next section.

The measurements reported in the plots presented in this chapter are taken on the test architecture described in Chapter 5. We decided to anticipate some results in order to illustrate better the behavior of the discussed aspects.

4.1 Sliding window partitioning

In this section we deal with the problem of how to partition the points arriving in a sliding window. We remark that the partitioning is did "on the fly", in the sense that for each incoming point a proper owner is selected.

We can make a first distinction between the partitioning schemes that are applicable to this problem.

- **Load unaware**. Do not take in account the actual load of the workers. The partitioning process follows a set of rules that are always the same, even if there is a non-negligible factor of load unbalancing.

- **Load aware**. The partitioning is based on the actual load of the workers. The main objective is to maintain the workers' load balanced.

The partitioning techniques differ also on if they depend or not from the spatial distribution of the received points: in the first case the data space is partitioned in according to a certain scheme and the resultant partitions are mapped on the workers. When a new point is received, the owner selection consists in finding the worker whose associated partition is the one in which the new point falls into. In the other case, the heuristics through which the owner selection is done are independent from the points' spatial coordinates.

The spatial distribution dependent partitioning techniques present two main issues:

1. if the points are not uniformly distributed in the data space, then the workers' load could be unbalanced;

2. the emitter cannot take any counter-measure against the load unbalancing since the partitioning is constrained by the points coordinates.

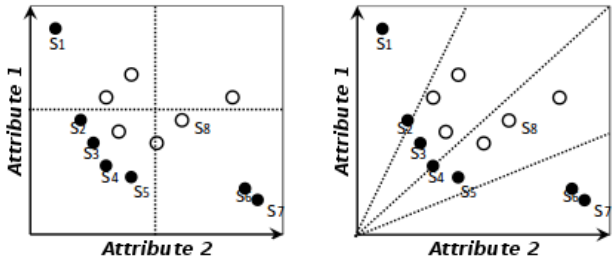
In what follows we discuss partitioning schemes that are found in literature, as the grid and angle-based partitioning, and propose a set of new techniques with the aim to better balance the workers' load.

4.1.1 Load unaware partitioning

In literature, one of the most prevalent methods employed for the space partitioning, in skyline processing, is the grid partitioning. Basically it consists in recursively splitting each dimension of the data space in two parts, dividing the data space in a set of hypercubes and assigning each of them to different workers.

When a new point arrives, the emitter has to individuate the hypercube in which the new point falls into, designing as owner the worker to the which that hypercube has been assigned.

Variants of this type of partitioning are found in both in distributed [23] and parallel [20] configurations. The main advantage of this approach is that, if there is a one-to-one mapping between partitions and workers, it is possible to establish dominance relations directly among the workers. For instance, consider the grid partitioning of a 2-dimensional data space, illustrated in 4.2a: if a point falls into the bottom left partition, then all the points falling in the top right partition will be dominated by it. In general, if the upper right corner of a partition a, dominates the lower-left corner of a partition b, then all the points contained in b are dominated by all the points of a.



(a) Grid partitioning (b) Angle partitioning

Figure 4.2: Example of 2-d partitioning. The skyline points are black filled.

The main drawbacks of this partitioning schema are:

- many partitions do not contribute to the skyline definition. As shown in 4.2a, most of the skyline points fall in the lower left partition, hence they are all assigned to the same worker.

- the number of points owned by each worker depends on their spatial distribution. Consider again the Figure 4.2a: it is clear that the worker associated with the upper right partition is less loaded than the worker associated with the bottom left one.

The authors of [2] deal with the problems introduced by the grid partitioning, proposing an angle-based partitioning scheme. An example of application of this method, in a 2-dimensional data space, is shown in Fig. 4.2b. This approach increase the efficiency of skyline query processing, since the skyline points are more fairly distributed among the partitions respect to the grid partitioning scheme. However, the number of points assigned to each partition, and thus to each worker, is still dependent from the points' spatial distribution.

A different approach to the partitioning problem is to make totally independent the partitioning scheme from the spatial distributions of the points. Consider a round-robin owner selection: the owner election does not depend on the spatial coordinates of the received points, but it is done in a round-robin fashion. Decoupling the owner selection from the space distribution allows to be fair from the view point of load distribution among the workers. However, even in this case, the load unbalancing introduced by the pruning is still present.

4.1.2 Load aware partitioning

The load aware partitioning schemes aim to try to maintain the load among the workers balanced. We remark that we have two source of load unbalancing: the one introduced by the pruning and the one introduced by the partitioning schema if it relies on spatial coordinates of the arriving points. All the heuristics presented in this section are spatial distribution independent, in order to restrict the load unbalancing sources to the only one introduced by the pruning.

Fastest worker

Basically it is an on-demand policy. An arriving point is sent in multicast to all the workers, the first which is able to accommodate the new point in its queue become the owner. The partitioning in made directly during the multicast did by the emitter. The pseudocode of the partitioning multicast is reported in Algorithm 7.

This heuristic is designed for low asynchrony degrees. In fact, from its view point, there are no differences between a channel which is almost full

Algorithm 7 Multicast(p). Fastest worker multicast pseudocode.

```
1: sent ← 0
2: i ← 0
3: while sent < N do
4:    if i-th channel's buffer is not full then
5:       if p has still not an owner then
6:          set worker i as the owner of p
7:       end if
8:       send p through the channel i
9:       sent ← sent + 1
10:   end if
11:   i ← i + 1  mod N
12: end while
```

and one which is totally empty, they have equal probability to be selected[1]. If the asynchrony degree is low, e.g. one, the difference between "almost full" and "totally empty" is about 1 element, hence the balancing is good. In other cases, when the asynchrony degree is higher, the same worker could be chosen as owner of new points, until its queue becomes full.

The experiments confirmed this behavior. In Fig. 4.3 we show, at each timestamp, the difference between the number of points owned by the most loaded worker and the number of points owned by the least loaded one. It is possible to observe that lower the asynchrony degree, lower the load difference, and so the load unbalancing. The best results is achieved with a queue length equal to one: in this case a worker is selected as the owner for a new point when it is effectively ready to process it. When the asynchrony degree is higher, instead, a worker could be designed as owner also in the case in which its queue is almost, but not totally, full.

In the optimal case, it is possible to note that there is an initial "transient" period where the load difference is high. This is due to the fact that, at the beginning, all the workers are empty, so the first worker will be selected until its load allows it to satisfy the stream arrival rate. When it will be no more able to sustain that rate, the heuristic will select another worker, until all of them will become bottleneck. At that time the system reaches a "steady-state" where the load difference between the workers remains low.

[1]To be rigorous: they have the equal probability only if, in Alg. 7, the starting point of the loop (i) is chosen at random

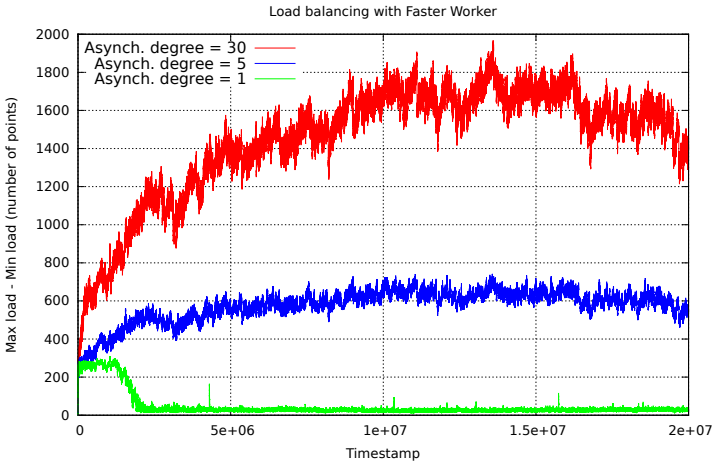


Figure 4.3: Fastest worker: workers' load difference in function of time, with different asynchrony degrees. Parallelism degree fixed to 10. Independent data distribution. Avg number of non-obsolete point per sliding window size equals to 4400.

Actual load

The idea is to give to the emitter the knowledge of the number of points currently owned by each worker. For each new point, the owner selection is done selecting the least loaded worker.

If the module is bottleneck, the asynchrony degree of the queues from the emitter to the workers, plays an important role. The information on which the emitter bases the owner selection do not take in account the points that are already in the workers' queue. Since the processing of each of these points can lead to variations in the workers load, we can state that higher the asynchrony degree, more outdated are the information on which the emitter bases its partitioning policy.

For each enqueued point, the actions impacting on the load, performed by each worker are:

- delete all the points dominated by the new one;
- store the new point if it is the owner.

The arriving of a new point can lead the worker to perform zero or one stores and zero, one or more deletes.

Consider the case in which, at time t, the emitter assigns a newly arrived point p to a worker W_i because, in according to the held information, at that

time it is the least loaded worker. That point will be received by its owner at time $t+\tau_i$, where τ_i is the time needed by W_i to serve all the points enqueued before p. When the point will be received by the designated worker, there are no guarantees that this last is still the least loaded worker, since all the points served in τ_i potentially modified its load. The term τ_i is defined as a function of:

a) the number of points enqueued before the new one;

b) the service time of the worker i.

If the module is a bottleneck, the queues between the emitter and the workers are full: the number of points enqueued before the new one is exactly the asynchrony degree. The consequence is that the error of this heuristic, in selecting the least loaded worker, is proportional to the asynchrony degree.

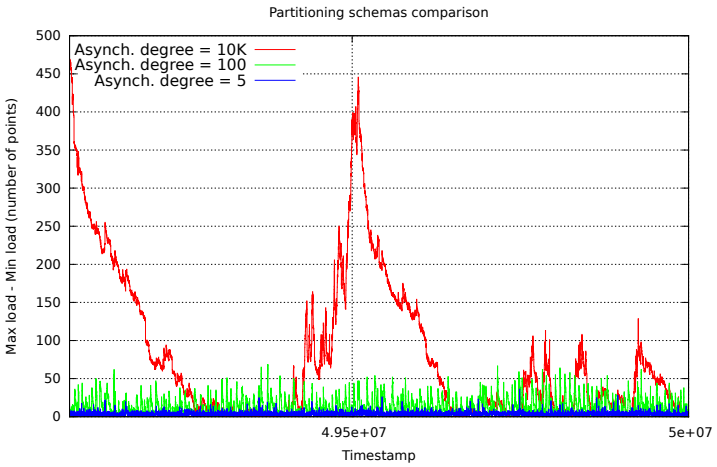


Figure 4.4: Actual load: workers' load difference in function of time, with different asynchrony degrees. Parallelism degree fixed to 10. Independent data distribution. Avg number of non-obsolete points per sliding window equals to 4400 points.

In Figure 4.4, the load difference between the most loaded worker and the least loaded one is reported. It is possible to observe that higher the asynchrony degree, higher the load unbalancing among the workers. It is worth noting that, with an asynchrony degree of $10K$, the peaks reached by *Actual load* are about one order of magnitude lower than *Fastest worker* with an asynchrony degree of 30.

Queue length

This heuristic is an extension of *Actual load*. It aims to reduce the error, introduced by this last, in the selection of the least loaded worker when the module is bottleneck and the asynchrony degree is high. The idea is that the emitter counts, for each worker, in addition to its actual load, also the number of the still enqueued points for which it has been designed as the owner.

In this way, assuming that the module is a bottleneck, if a point is received by the emitter at time t, the owner selection will not be based only on the workers' load at time t, but it will take in account also the additional load that each worker will have at time $t + \tau_i$.

Clearly, also in this case, we have an approximation, since we do not consider the pruning that each enqueued point could cause. Anyway, the error will be lower respect the one introduced by *Actual load*, due to the fact that we try to forecast what will be the load of the workers at time $t + \tau_i$ if we assign to it a point at time t.

Please note that, in the case in which the module is not a bottleneck, this heuristic is equal to the one presented in Section 4.1.2.

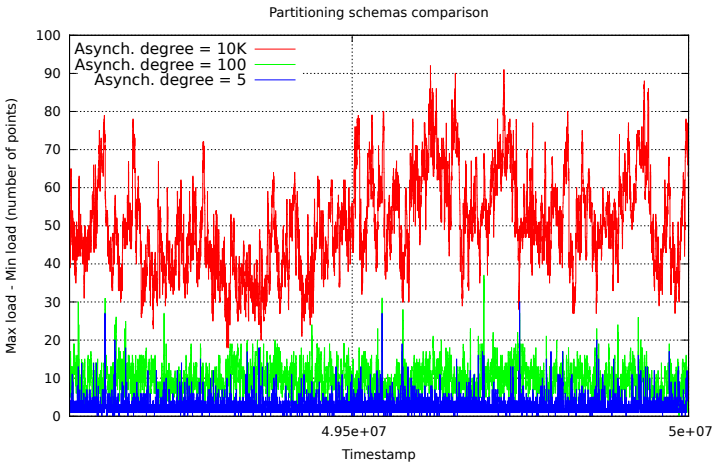


Figure 4.5: Queue length: workers' load difference in function of time, with different asynchrony degrees. Parallelism degree fixed to 10. Independent data distribution. Avg number of non-obsolete points per sliding window equals to 4400.

In Figure 4.5 the load difference between the most loaded worker and the least loaded one is reported. It is possible to observe that this heuristic

works better than the *Actual load*. The workers' load forecasting allows the emitter to select, with higher probability, which will be the least loaded worker at time $t + \tau_i$ where, as usual, the term τ_i is time needed by the worker W_i to serve a point submitted to it at time t.

Discussion

If we consider lower values of the asynchrony degree, the best result is achieved by *Queue length*, as shown in Figure 4.6.

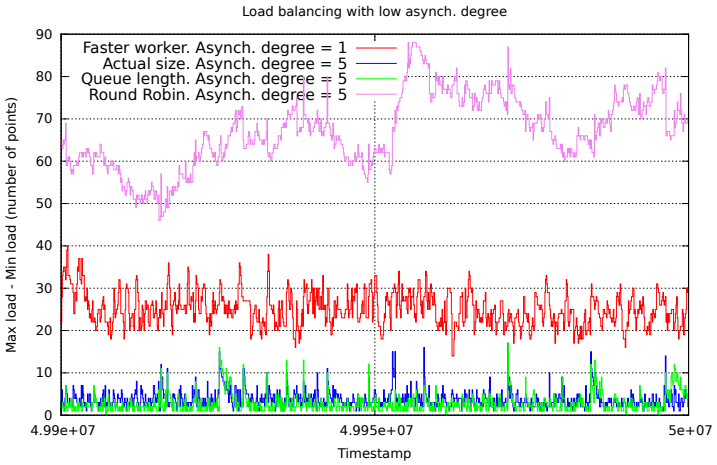


Figure 4.6: Worker's load difference in function of time, with different asynchrony degrees. Parallelism degree fixed to 10. Avg number of non-obsolete points per sliding window equals to 4400 points.

In the comparison we consider the best result of *Fastest worker*, the one achieved with a queue length of one, since in the other cases this heuristic introduces a load unbalancing which is not comparable with the results of the others.

As expected, *Actual load* and *Queue length*, are very similar in terms of load unbalancing when the asynchrony degree is low. We remark that the only difference between the two is that the latter takes in account the presence of a queue between the emitter and each worker, trying to forecast what will be the load of each worker when a point, eventually enqueued at the current time, will be served. If the queue length is small, the difference between the two approaches is negligible.

If we consider higher values of the asynchrony degree, the situation is the one reported in Fig. 4.7. The measurements on *Fastest Worker* are not shown, since it is not designed for high values of asynchrony degree.

Also in this case, *Queue length* performs better than all the others. As confirmed by the data reported in Table 4.1, it is the one which is able to maintain the lowest mean load difference and variance.

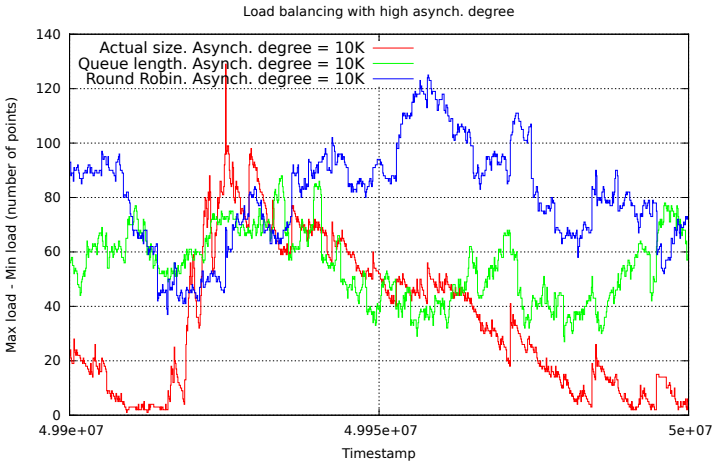


Figure 4.7: Worker's load difference in function of time, with different asynchrony degrees. Parallelism degree fixed to 10. Avg number of non-obsolete points per sliding window equals to 4400 points.

Asynch. deg.	Round Robin		Fastest worker		Actual load		Queue length	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
1	-	-	34.13	1790.67	-	-	-	-
5	64.20	230.88	581.45	3948.31	3.64	4.60	2.61	4.33
100	64.54	246.26	-	-	12.93	136.26	10.89	23.06
10K	70.06	424.65	-	-	104.60	16483.29	50.90	280.42

Table 4.1: Load difference between the most loaded worker and the least loaded one.

In conclusion, the *Queue length* heuristic is the one which is able to reach the best results in terms of load balancing among the workers, even if the asynchrony degree is high. It is worth noting that an higher asynchrony degree allows the module to sustain an input stream with an higher variable arrival rate, without blocking. If we want to employ *Fastest worker* as partitioning schema, we have to maintain an asynchrony degree equal to one, otherwise the heuristic does not works.

In this section the presented partitioning schemes have been analyzed only from the point of view of the load balancing. However, in the Chapter 5, a comparison among them from the view point of their impact on the module service time, is reported.

4.2 How it works

This section describes how the parallel module is organized. The sequential computation is analyzed from the view point of the operation that each worker can complete locally to itself and the ones requiring a global cooperation. Moreover, the behavior of the emitter and the collector nodes is explained.

As said before, the module is activated in two cases: when it receives a new point and when an event of its event list is raised.

In the former case, the sequential module performs the following steps:

1. pruning: discard all the points dominated by p.
2. store: store p into the employed data-structure.
3. SIT: calculate SIT_p and insert the proper event in the event list.

In the case in which the module timer reaches the time of the oldest event in the event list, the sequential module is internally activated:

1. if the raised event is an expire event, delete the point associated with it.
2. emit a proper skyline update and update the event list according to the event's type.

The activation behavior is replicated over the workers. Since the emitter multicasts an arriving point to all the workers, for each incoming point all the workers will be externally activated. Differently, since each worker stores a disjoint partition of the internal state and the events are relative to the stored points, the workers can be internally activated independently from the others.

The sequential module has an internal timer used for: a) timestamp the newly arrived points; b) mark the time in order to raise the events of *skytime* or *expire*. This behavior is reproduced on the parallel module as follows:

- the emitter and all the workers have their own timers, they are not necessarily synchronized.
- the emitter is in charge to timestamp the newly arrived points.
- for each received point, each worker resets its timer to the point arrival timestamp (the one set by the emitter).

If this last operation is not performed and the module is bottleneck, the points can waste time during which they should be part of the skyline into the queue of their owners. Consider the limit case in which a point arrives to the emitter at time t, its expire time will be $t_{exp} = t + W$. Since the module is bottleneck, the point will be server by its owner at time $t + \tau$ and, if $\tau \geq W$, it will be served when it will be already expired, preventing de facto its entering into the skyline.

4.2.1 Emitter

When a new point p is received, the emitter performs the following actions:

1. select the worker which will be responsible for p (the owner of p);
2. timestamp p
3. send p to all the workers

The owner selection is done in according to the employed partitioning scheme.

4.2.2 Worker

The target is to replicate the behavior of the sequential SIT among a set of workers in a way such that the semantic of the original algorithm is respected. We have to consider that the internal state is partitioned among the workers, and each worker is responsible for the points in its partition.

Consider the computation executed at each point arrival:

1. the pruning phase is local to the partition of each worker. Since the new point is sent to all the workers, each of them can perform the pruning locally to its partition;
2. the store operation must not be executed by all the workers, but only by the one to which the new point has been assigned;
3. the SIT of the new point must be computed as the expire time of its critical dominator. The problem of finding the SIT of the new point can be redefined as the problem of finding the youngest dominator among all the points dominating the new one (the critical dominator). Since the dominators of the new point can be distributed among all the workers, the research of the critical dominator is an operation involving them all.

SIT calculation The skyline influence time of a new point p is determined through a reduce phase. When a point p is received by a worker, the local critical dominator of p is calculated. It is worth noting that this result is local to the partition of the worker. At the end of this phase, each worker W_i has calculated its local SIT of p, let us call it SIT_p^i, as the expire time of the local critical dominator of p.

Once each W_i has calculated its SIT_p^i, the calculation of the "global" SIT of p must be done. For this purpose, all the workers participate to the reduce phase, in order to compute:

$$SIT_p = \max_i SIT_p^i$$

The reduce result is sent to the owner of p, allowing it to create and insert the proper event in its event list (as discussed in 3.2.1).

Internal activation Each worker manages its own event list. As in the sequential SIT, an event is described by a triple (p, t, tag) where p is a point, t is time at which the event must be raised and tag indicates the type of the event (*skytime* or *expire*). The events in the event list of a worker are relative to only the points owned by it.

Algorithm 8 Worker pseudocode. *w_time* is the worker's internal timer.

```
1:  w_time ← 0
2:  while true do
3:    if there is a point to receive then
4:      receive the point p
5:      w_time ← p.t_arr
6:      dataProcessing(p)
7:    else
8:      increment w_time
9:    end if
10:   dataMaintenance(w_time)
11: end while
```

In Algorithm 8, the pseudocode of the generic worker is reported. It is possible to observe how a worker is activated: if a new point is received, then the worker is externally activated (lines 4-6). In this case the worker time (w_time) is reset to the new point arrival timestamp and the new point is processed. If there are no points to receive, the worker increase its time in order to activate itself when the timestamp of the oldest event will be reached (through the function $dataMaintenance()$).

4.2.3 Collector

All the workers send skyline updates concurrently. As said in Section 2.3, the module must guarantee the output chronological order. If the collector simply forwards the skyline updates received by workers, since there are no guarantees that the workers are time-synchronized, the output chronological order could not be preserved. Due to this motivations, the collector is designed in way such that it is able to order the skyline updates received by the workers.

The collector maintains the least common timestamp among all the workers, let us call it t_{coll}. The updates with a time greater than t_{coll} are bufferized until all the workers will have reached that time. The bufferized updates are time-ordered. When all the workers reach a new common timestamp, the t_{coll} is updated and the buffer is empty till the new t_{coll}.

4.3 Data structures

As in the sequential module, all the non-obsolete points of the current sliding window must be kept in memory. In the sequential module, the operations performed on the indexing data structure are the following:

- insert/delete points. An insert operation is executed for each point arrival. A delete operation is executed for each point expiration and for each pruned point.
- given a point, search all the points dominated by it. This operation is necessary to perform the pruning operation at each point arrival.
- given a point, search its critical dominator. For each newly arrived point, its critical dominator must be found.

In the parallel module, each worker has its own data structure in which the owned points will be stored. It is clear that, if there are no load unbalancing problems, the number of points stored in each worker will be $\frac{K}{n}$ where K is the number of non-obsolete points in the current sliding window and n is the parallelism degree.

4.3.1 R-Tree vs Vector

The R-Tree [1] is an height-balanced tree used for indexing multi-dimensional objects. In literature, there are several variants of this data structure. In this thesis we consider the R*-Tree [18], which is optimized for the indexing of multi-dimensional points. As discussed in [26], [27] the R-Trees do not have provable performance guarantees, but it is well known that they present reasonably performances for real-world data.

This type of data structure was proposed as indexing structure in a number of works concerning skyline queries, both for sequential evaluation over a data stream [26], [25] and sequential/parallel evaluation on static datasets [8], [2], [21], [22].

The R-Tree is a self-balancing data structure: it ensures to maintain the number of childs that each node will have, in a range which is an user-specified parameter. When, due to an insertion or deletion, there are one or more nodes with a number of childs not belonging to the specified range, a variant-specific algorithm is executed in order to re-balance the structure. This characteristics ensures that the height of the tree is always logarithmic in the number of stored element. However, when the rate of inserts/deletes operations is high, such in our application, the re-balancing costs could not be negligible.

R-tree size Since the R-Tree is a tree data structure, we expect that operations, like as searching, will have a logarithmic behavior with additional amortized costs deriving from re-balancing routines. However, as known, the logarithmic costs of tree data structures are asymptotic: there are hidden constants that are negligible only when the number of elements in the tree is high. In this regard, with the employed parallelization schema the following aspects must be considered:

- assuming that there is no load-unbalancing, the number of non-obsolete points in the current sliding window, which are the ones to be stored, is equally distributed among the workers;
- the pruning operations can drastically reduce the number of points stored by the workers.

Even with an high number of points received in a sliding window, the number of points stored per-worker could be one or more order of magnitude lower, as shown at the end of this section, preventing the R-Trees to reach their asymptotic performances.

Vector Considering the above mentioned aspects, we decided to compare the R*-Tree-based implementation with a vector-based one. Each worker stores its own points into a vector and, for each received point, the vector is scanned in order to find: a) the points dominated by the new one, they are the ones to be pruned; b) the points dominating the new one for establishing its local critical dominator, and hence its local SIT. Since the order in which the points are stored is not important, the insert is done at the end of the vector in constant time. The delete is done exchanging the last element with the one to be deleted and finally deleting it. In such a way the delete operation does not create holes and it can be performed also in constant time.

The points are stored using an attribute-oriented organization: d separated vectors are used to contiguously store the same coordinate of different points. Using this approach the operations performed in the scan loop are ready to be vectorized and the memory access pattern allows a better use of the cache hierarchies.

Comparison In Figure 4.8 a comparison between the version with the R*-Tree and the one with the vector is shown. The plot reports the sequential module service time as a function of the number of the non-obsolete points per sliding window.

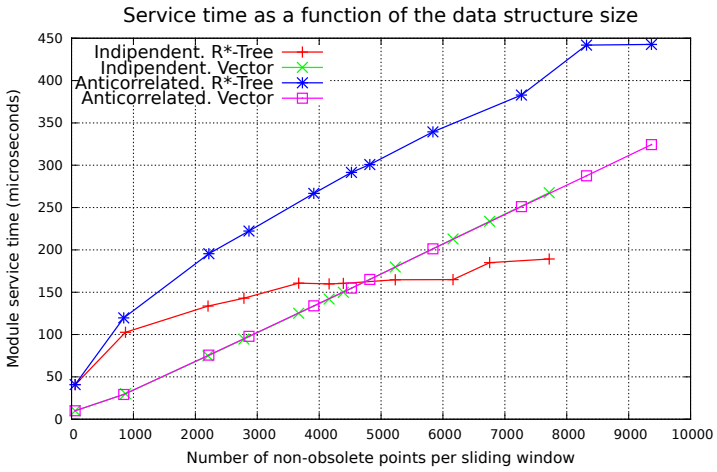


Figure 4.8: R-Tree vs Vector comparison. It is the module service time in function of the data structure size.

As expected, the service time of the vector-based implementation is totally independent from the spatial distribution of the points: both in the case of independent and anticorrelated distribution, the service time grows linearly with data structure size. As far as R*-Tree based implementation is concerned, it can be noted that also the data spatial distribution influences the service time. In both the independent and anticorrelated cases, the service time has a sub-linear growth, but the former case presents a lower multiplicative constant respect to the latter one. We can state that this multiplicative constant depends on the spatial distribution of the points.

We want to stress the point that the measurements of the above graph are relative to the sequential module. In the parallel version we have to consider that the number of non-obsolete points will be partitioned among the workers. Moreover, it is worth noting that the number of non-obsolete points is not equal to the number of points received in a sliding window. This last can be expressed as the product between the stream arrival rate (tuples/seconds) and the sliding window temporal length (seconds). The number of non-obsolete points, instead, is the number of points received in a sliding window survived to the pruning caused by the subsequent arrival of other points.

In Figure 4.9, the number of non-obsolete point in function the number of arrived points in a sliding window is shown, both for the independent and anticorrelated distributions. It is interesting to observe that, even in the anticorrelated case (the one with the lowest pruning activity), the number

of non-obsolete points is at least two order of magnitude lower than the total number of points received in a sliding window.

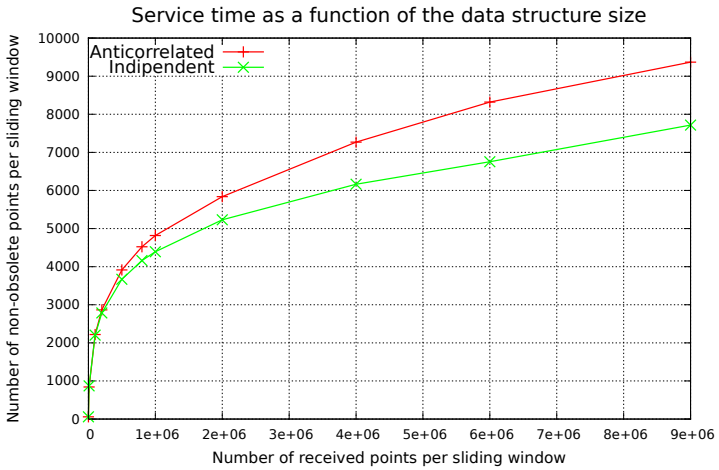


Figure 4.9: Number of non-obsolete points in function of the number of arrived points per sliding window, and data spatial distribution.

According to [25], in most sliding window applications, we may expect a total number of points received in a sliding window less or equal than 10^6. It is possible to observe that, for a number of non-obsolete points corresponding to a number of points received in a sliding window equals to 10^6, the vector-based implementation still outperforms the R-Tree-based one, even for the independent data distribution. Due to this observation and the independence of the vector-based version from the data spatial distribution, in what follows we will discuss only about this last one.

Theoretical discussion In addition to what discussed above, we can make some considerations on the ideal scalability of the the R-Tree-based implementation and, in general, of a module whose service time can be expressed as:

$$T_S = F(M) = T_F * log(M)$$

where T_f is a multiplicative constant and M is the size of a data-structure over which F is applied. In our specific case, M is the number of points stored in the R*-Tree.

Consider the data-parallel version of this module in which the load is partitioned among the workers, the ideal service time with a parallel degree

of n is:

$$T_{id}^{(n)} = T_f * log(\frac{M}{n})$$

leading to an ideal scalability of:

$$s^{(n)} = \frac{log(M)}{log(M) - log(n)}$$

This shows us that, from a theoretical view point, we cannot expect a linear scalability from the data-parallelization of a module whose service time can be expressed as logarithmic function of its load. This is another motivation of our choice to use a vector as the data-structure of our module.

4.4 Asynchronous reduce

The skyline influence time of each new point is calculated through a reduce phase involving all the workers. The SIT of a point is useful only to the owner of that point. In fact, the SIT of a point is used for the creation of its relative event (*skytime* or *expire*, discussed in 3.2.1), which is an operation done by its owner.

For each point, all the workers participate to the reduce phase but, if the reduce is synchronous, only the owner have to wait its conclusion. The time necessary to the completion of the reduce is the reduce latency. That latency is influenced also from load unbalancing situations in which the owners wait time will be dominated by the service time of the slowest worker.

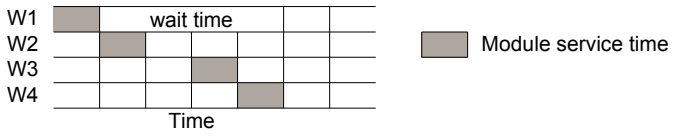


Figure 4.10: Synchronous reduce for the SIT calculation of a point assigned to the worker W1. It must wait until all the participation of all the others is completed.

Moreover, if the module is bottleneck, different workers could serve the same point at different times, as illustrated in Figure 4.10. In that example, W1 is the owner of a point which is served by the various workers in the time slot drawn in gray. The worker W1 have to wait until all the others participate to the reduce phase of that point. Clearly this waste of time impacts on the workers' service time and, if the partition schema is load unaware, also on the load balancing.

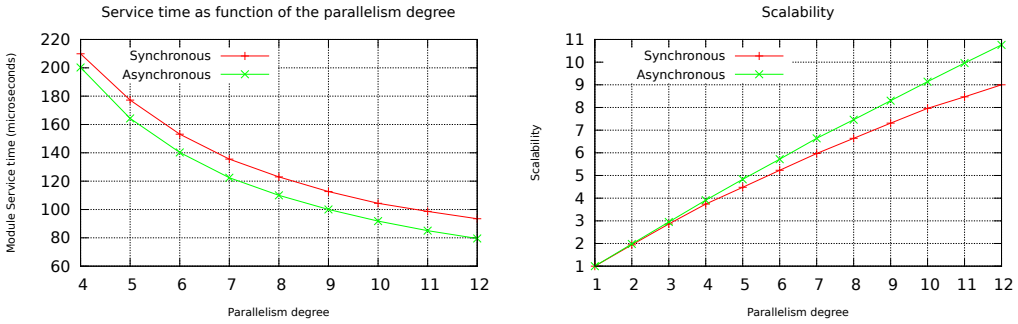
The idea is to make the reduce asynchronous: the owner of a point has not to wait the completion of the reduce to process all the subsequent points: it is like an out-of-order processing. Such an approach is meaningful only if the processing of the other points does not depend on the point for which the reduce is still incomplete, let us call it the pending one.

Consider a worker with one or more pending points, the subsequent points interact with them in the following ways:

- the local critical dominator of the new points must be calculated: the information needed by this operations are the expire time of the stored points and their spatial coordinates. The SIT of the older points is not involved, so this operation is independent from the reduce result;

- all the points dominated by the new ones must be pruned: the pruning is based only on the spatial coordinates of the points. Even in this case the reduce result is not involved.

We can conclude that the point's processing phase does not depends on the SIT of the others points. The asynchronous reduce works as follows: when a point is received by a worker, whether it is the owner or not, it participates to the reduce phase without waiting for the result. When the reduce will be complete, its result will be sent to the owner of the point to which the reduce is referred. All the workers non-deterministically wait for the reception of new points from the emitter or for the result of a reduce. If the result of a reduce relative to a point p is received, the proper event associated with p is inserted in the worker's event list. As in the sequential case, the event type is established in according to the received SIT_p.



(a) Service time

(b) Scalability

Figure 4.11: Comparison between the synchronous and asynchronous reduce. In (a) the module service in function of the parallelism degree. In (b) the scalability.

In Figure 4.11, the comparison between the implementation with and without the discussed optimization is reported. As can be noted, the higher the parallelism degree, the higher are the benefits from the adoption of the asynchronous reduce. This behavior is motivated by the fact that the higher the parallelism degree, more difficult is the balancing of the load with the consequence that, in the synchronous reduce case, the owners are blocked for longer time due to the increase of the reduce's latency.

4.5 Maximum sustainable rate

In this section the proposed parallelization is analyzed from the view point of the maximum reachable bandwidth. With this term we mean the maximum arrival rate of the elements of the input stream which the module is able to sustain before it becomes bottleneck, given a certain parallelism degree.

Consider the hypothetical case in which no pruning operations are performed. In this case, the internal state of the module correspond to all the points received in a sliding window, let us define its size as $K_{no\_pruning}$:

$$K_{no\_pruning} = \lambda \times W$$

where λ is the stream arrival rate and W is the sliding window length.

Introducing the pruning, each point has a certain probability to be pruned, let us call it p. It is the probability that a point is pruned due the subsequent arrival of a point dominating it, in the same sliding window. We can model the number of non-obsolete point in a sliding window as:

$$K = \lambda \times W \times (1 - p) = K_{no\_pruning} \times (1 - p)$$

The sequential module service time can be expressed as:

$$T_S = F(K)$$

where the function F is defined by the considered implementation of the module (R*-Tree-based or the vector-based). In the case of the vector-based implementation, the function F is defined as

$$F(K) = T_F * K$$

where T_F is a constant. Since we consider the implementation with the asynchronous reduce, this evaluation do not take in account the time needed by the reduce phase.

In according with the methodologies discussed in [16], the module is a bottleneck if:

$$T_A = \frac{1}{\lambda} > T_S$$

where T_A is the inter-arrival time. In this case we can find an optimal parallel degree $\bar{n}$, such that:

$$T_S^{(\bar{n})} = \frac{T_S}{\bar{n}} = T_A$$

Combining the above definitions, we can express the optimal parallelism degree, i.e. the one for which the module is not a bottleneck given λ and W, as:

$$\bar{n} = \lambda^2 \times W \times (1 - p) \times T_F$$

Given a parallelism degree n, the module is not a bottleneck for:

$$\lambda \leq \sqrt{\frac{n}{W \times (1 - p) \times T_F}}$$

The pruning probability p depends on: a) the number of arrived points in a sliding window, i.e. $\lambda \times W$; b) the spatial distribution of the incoming points.

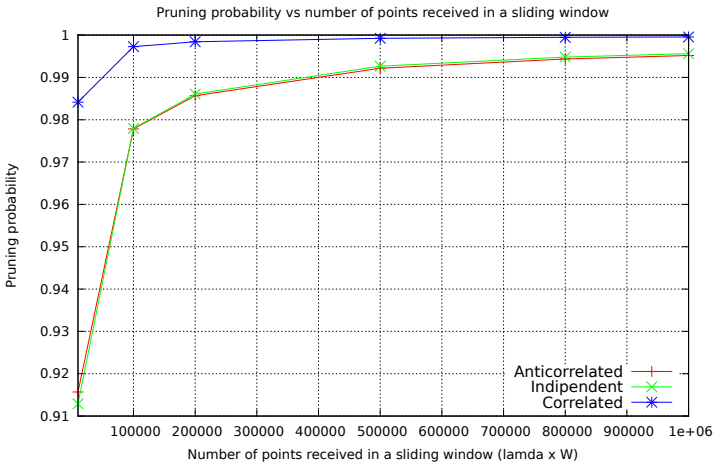


Figure 4.12: Pruning probability in function of the number of points received in a sliding window.

The Figure 4.12 confirms what said above. The higher the number of points received in a sliding window, the higher the pruning probability. As far as concerning the spatial distribution, as discussed in 2.4, the anticorrelated distribution is the one leading to the lowest pruning probability while the correlated one leads to highest p. The independent data distribution is similar to the anticorrelated one in terms of pruning activity.

Chapter 5

Experiments

As explained in Chapter 4, if the module is a bottleneck, the sequence of the skyline updates produced by the module is correlated to the utilization factor of this last. To ensure the repeatability of the tests the module is implemented in way such that the sequence of the skyline updates and the utilization factor are decoupled. The module expects to receive points in the following format:

$$p = (d_1, d_2, ..., d_n, t_{arr})$$

where t_{arr} is the arrival time of p. In such a way, the module has not to timestamp the arriving tuples, it must only reset its timer to the newly arrived point's timestamp. If the module is bottleneck and hence a point p arrived at time t_{arr} is served at time $t_{arr} + \tau$, where τ is the point's waiting time, the eventual updates relative to it will be emitted with time t_{arr} since the module timer will was reset to this time.

Given the rate at which the points must arrive to the module, this are generated in a way such that the arrival rate is constant. The generated points are stored in a plain-text file. A generator node has been implemented with the function of reading the points from a file and of emulating the input stream in according to the timestamp specified for each one of these. Moreover, a consumer node is implemented in order to catch the updates emitted by the module and verify if they describe a skyline or not. This last functionality can be interpreted as "on the fly" correctness check.

In this section we consider the "best" version of the proposed parallel module: the one with the asynchronous reduce optimization and adopting the vector as data structure for storing the non-obsolete points.

Test platform The experiments are executed on a commodity Intel multi-processor architecture composed by two Xeon E5-2650 CPUs for a total of

16 cores and 32 hardware SMT contexts (Simultaneous Multi-Threading). Each core is equipped with an L1 cache of 32KB and an L2 cache of 256KB. Groups of 8 cores share L3 caches of 20MB. The communication channels are implemented through efficient lock-free queues made available by the FastFlow library [14].

5.1 Sequential algorithms comparison

This set of experiments compare the two sequential algorithms presented in [26]. The target is to demonstrate that *SIT* outperforms *Lazy* both from the view point of the service time and memory occupancy. All the tests discussed in this sections are expressed as function of the *timestamp*. At each reported timestamp is associated one of the possible events:

- a point arrival. The timestamp indicates the point's arrival time;
- a point expire. The timestamp indicates the point's expiration time;
- both an arrival and an expire;

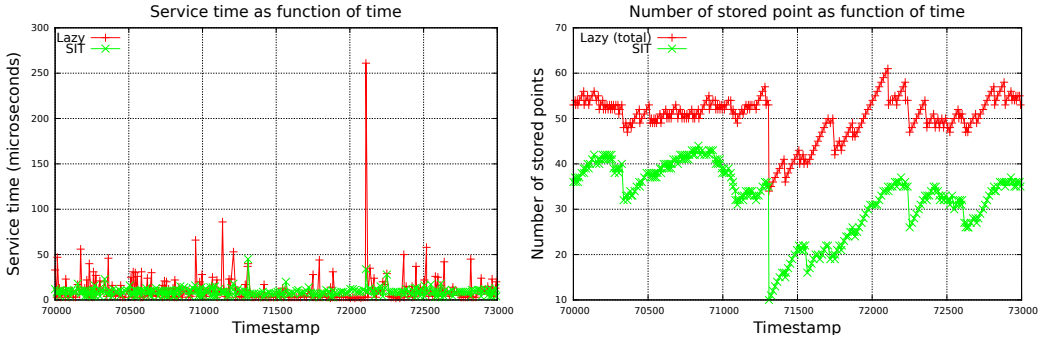
We will analyze only a snapshot of the entire stream computation, in order to illustrate better its details. Moreover, in all the following experiments, the number of dimensions is fixed to 4, the stream arrival rate to 10K tuples/second and the sliding window length to 3 seconds.

For each data distribution (discussed in 2.4), three types of measurements are shown: a) the module service time. It corresponds to the time necessary to handle one of the above described events; b) the total number of stored points by the two algorithms. This test is useful to compare them in terms of memory occupancy; c) relative to only the Lazy algorithm. It compares the number of points stored in DB_{sky} and DB_{rest}.

5.1.1 Anticorrelated distribution

In Fig. 5.1a the service time for each timestamp is reported. It is possible to observe the different behavior of the two algorithms. Lazy, for each point arrival, checks only if it is skyline or not, inserting it in the proper data structure. When a point expires, the *maintenance* function is executed: all the points exclusively dominated by the expiring one are inserted into the skyline. The spikes presented by Lazy correspond to the execution of the maintenance function. The behavior of SIT is different: since at each point arrival the *skyline influence time* of the new point must be calculated, the

data processing function is more costly. On the other hand, the computational cost of the maintenance function is negligible. It is clear that the Lazy approach leads to an higher variance of the service time, if compared with SIT. Due to this aspect, an hypothetical data-parallel version of Lazy could be characterized by an high variance in the service time of the workers: this could have a negative impact on the parallel module service time.



(a) Service time (b) Stored points

Figure 5.1: Lazy vs SIT. Service time (a) and data structures size (b) as function of time.

In Fig. 5.1b the comparison between the total number of points stored by Lazy and SIT is shown. It is possible to observe that SIT is capable to keep a lower number of stored points than Lazy, since it performs pruning over all its DB. As an example, consider the timestamp 71300 in Fig. 5.1b: both the two algorithms discard a number of points due to the arrival of a point which dominates them. It is possible to note that the number of points discarded by SIT is higher respect to the ones discarded by Lazy. This is explained by the fact that this last performs pruning only over DB_{sky}, discarding the obsolete points in DB_{rest} only at their expiration time. However, due to the anticorrelated data distribution, which leads to a minimal pruning activity, this case is the one in which the gain of SIT respect to Lazy, in terms of memory occupancy, is the lowest, if compared with the other data distributions.

In Fig. 5.2 the comparison between the number of points stored in the two data structures employed by Lazy is shown. It is possible to observe the effects of the data distribution: the anticorrelated distribution leads to an high number of skyline points, which is comparable to the number of non-skyline points stored in DB_{rest}. An interesting aspect can be noted at timestamp 71300, where the newly arrived point leads to the pruning of a number of skyline points: the size of DB_{sky} is drastically reduced. However,

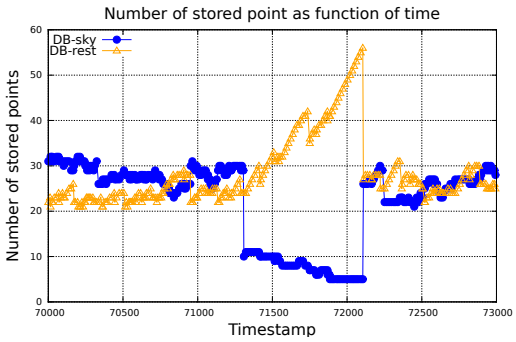
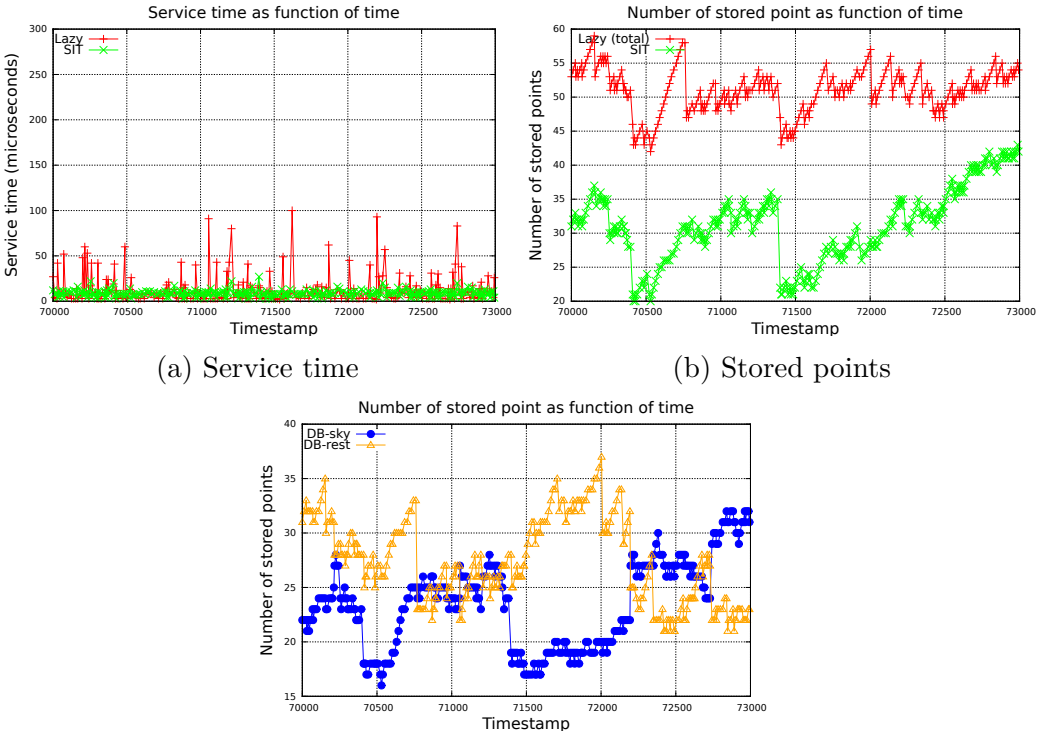


Figure 5.2: Lazy. Data structure comparison.

the size of DB_{rest} still grows, even if it contains some points having no more chances to enter into the skyline. At timestamp 72110 a skyline point dominating an high number of points expires: at that time all the new skyline points, i.e. the ones exclusively dominated by the expiring one, are moved from DB_{rest} to DB_{sky} and the DB_{rest} is cleaned-up, removing all the expired points.

5.1.2 Independent distribution

The independent distribution is characterized by an high number of both skyline and non-skyline points. If compared with the anticorrelated one, it presents a lower number of skyline point and an higher number of non-skyline ones. This aspect is also confirmed by Fig. 5.3c which shows that the number of points stored in DB_{rest} is higher, on average, than the ones stored in DB_{sky}. As a consequence, the module service time, reported in Fig. 5.3a, is higher respect to the anticorrelated case.



(a) Service time

(b) Stored points

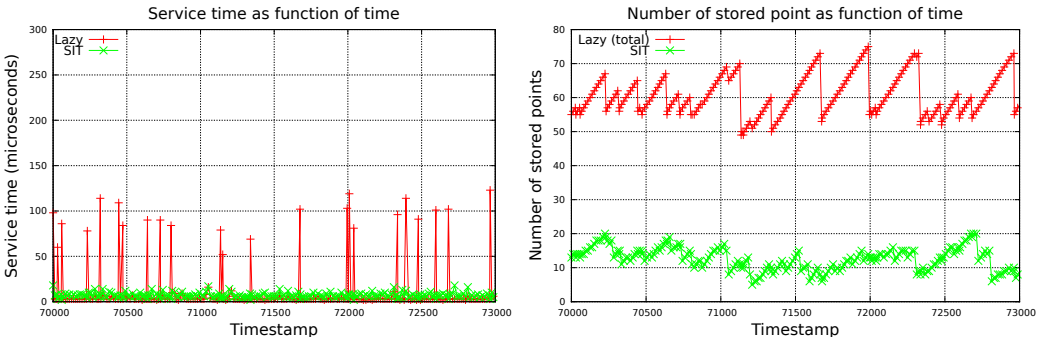
(c) Lazy. Data structure comparison.

Figure 5.3: Lazy vs SIT. Service time (a) and data structures size (b) as function of time.

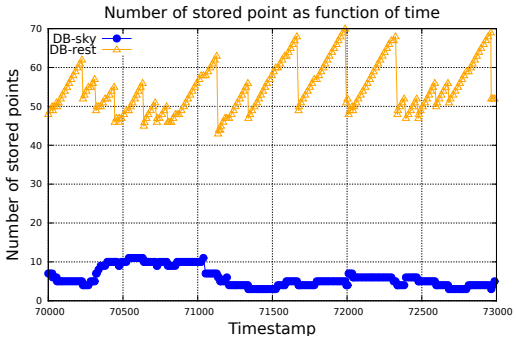
In Fig. 5.3b the number of points kept in memory, at each timestamp, by the two algorithms is shown. Differently from the anticorrelated case, the difference is more evident. As usual, this behavior is explained by the fact that Lazy does not immediately prune the obsolete points.

5.1.3 Correlated distribution

In the correlated distribution a small number of skyline points dominate all the others. The Fig. 5.4c shows this behavior: the number of points stored in DB_{sky} is much lower respect to the ones stored in DB_{rest}. The higher size of DB_{rest} leads to an higher cost of the Lazy maintenance function, as denoted in the Fig. 5.4a where the service time of Lazy reaches the highest peaks if compared with the other distributions.



(a) Service time (b) Stored points



(c) Lazy. Data structure comparison.

Figure 5.4: Lazy vs SIT. Service time (a) and data structures size (b) as function of time.

This kind of distribution leads to an intensive pruning activity by SIT: as shown in Fig. 5.4b, the difference between the total number of stored points by the two algorithms is the highest among all the three data distributions.

5.2 Effects of number of points per sliding window

This experiments show how the sliding window length and the input stream arrival rate influence the parallel module service time and hence its scalability. The sequential module service time is expressed as:

$$T_S = T_F \times W \times \lambda \times (1 - p)$$

where T_F is a constant, W is the sliding window length (in seconds), λ is the stream arrival rate and p is the pruning probability.

In according to [17], the parallel module ideal service time is

$$T_S^{(n)} = \frac{T_S}{n}$$

where n is the parallelism degree. The ideal service time is reachable only in the case in which the load is perfectly balanced among the workers. The ideal bandwidth is the inverse of the ideal service time. From now on, we refer the ideal bandwidth as the offered bandwidth, intending the bandwidth at which the module is able to consume the input stream. The required bandwidth, instead, is the stream arrival rate: if the offered bandwidth is greater than or equal to the required one, the module is not a bottleneck.

All the measurements presented in this section are taken fixing:

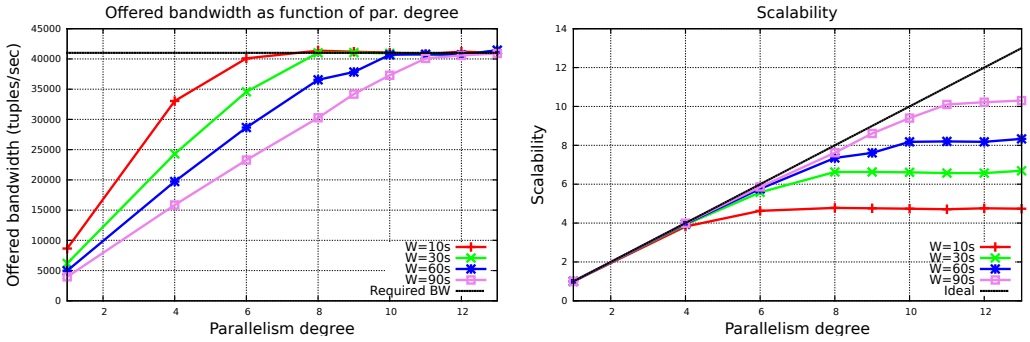
- the number of dimensions, equals to 5;
- the partitioning scheme, set to *Queue length*. As discussed in 4.1.2, it is the one able to maintain the load unbalancing among the workers to the minimum levels, if compared with the other discussed heuristics.

In according to [2], [7], this set of experiments is relative only to the anticorrelated and independent distribution. The correlated one is not considered since it leads to a very small number of non-obsolete points for the selected parameters: the module is never bottleneck and hence the scalability is meaningless.

5.2.1 Varying the sliding window length

In this experiment the module receives a stream of points with a fixed arrival rate and following different spatial distributions. For each data distribution different lengths of the sliding window are adopted. The target is to observe how the length of the sliding window impacts on the module offered bandwidth and hence on the scalability.

Anticorrelated distribution The Fig. 5.5a shows the offered bandwidth as a function of the parallelism degree, with different lengths of the sliding window. Moreover, the required bandwidth, i.e. the input stream arrival rate, is reported. In all the cases the module is able to solve the bottleneck with the available number of cores.

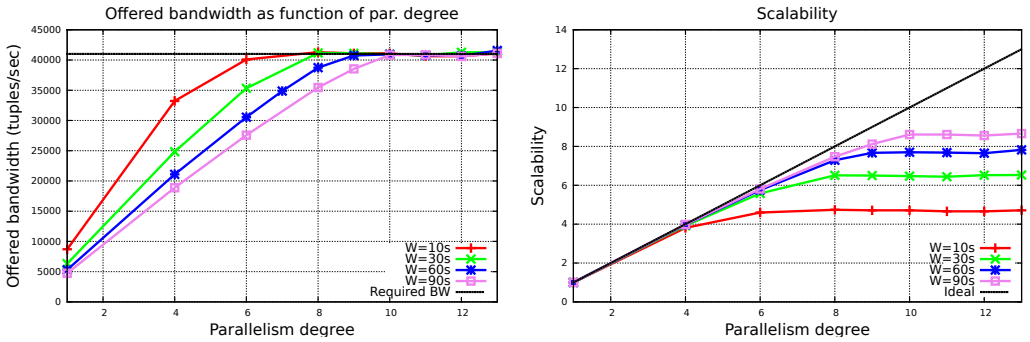


(a) Offered bandwidth (b) Scalability

Figure 5.5: Effects of the sliding window length on the service time (a) and scalability (b) as function of par. degree. Anticorrelated distribution. Input stream arrival rate: ∼40K tuples/sec

As expected, higher the window length, lower the offered bandwidth: the number of non-obsolete points increases with the growing of the window length, hence also the service time increases, since it is a function of the number of stored points. Please note, in Fig. 5.7, that the scalability curves stop growing after a certain parallelism degree: this is because the module, with that number of cores, is able to sustain the input stream arrival rate.

Independent distribution The independent distributions is characterized by an higher pruning activity than the anticorrelated one. Fixed a stream arrival rate and a window length, an higher pruning activity leads to a lower number of non-obsolete points and hence to a lower service time. This behavior is reported in Fig. 5.6a where, given a window length, it is possible to observe that the module has an higher offered bandwidth than the one of the anticorrelated case, specially with coarser grains, such as $W = 60s$ and $W = 90s$. The consequence is that the module is able to sustain the stream arrival rate, and hence not to be a bottleneck, even with lower values of the parallelism degree.



(a) Offered bandwidth (b) Scalability

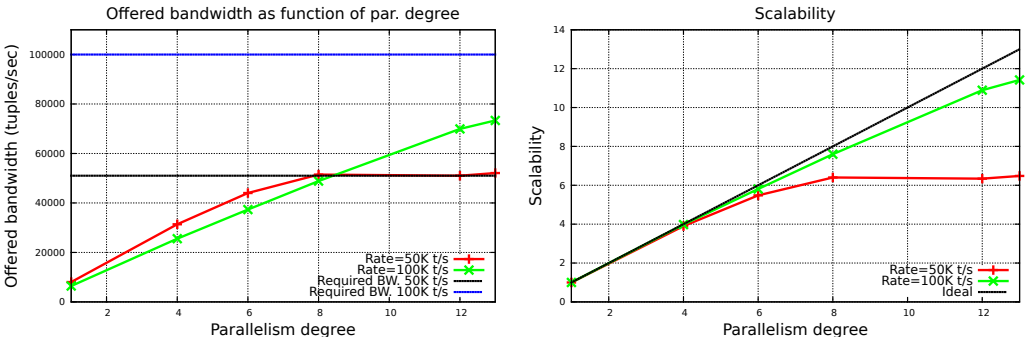
Figure 5.6: Effects of the sliding window length on the service time (a) and scalability (b) as function of par. degree. Independent distribution. Input stream arrival rate: ∼40K tuples/sec

It is worth noting that an higher pruning activity is source of load unbalancing. However, thanks to the employed partitioning scheme, this is reduced to negligible level, as confirmed by the scalability curves in Fig. 5.6b.

5.2.2 Varying input stream rate

This experiment is the counter-part of the previous one: the module offered bandwidth and the scalability are analyzed fixing a window length and varying the input stream arrival rate.

Anticorrelated distribution In Figure 5.7b the module is not bottleneck for a stream with arrival rate of $50K$ tuples/second with a parallelism degree of eight. With a rate of $100K$ tuples/second the bottleneck cannot be solved with the employed architecture. Considering the parallelism degree interval $[0, 7]$, where the module is a bottleneck even with a input stream at $50k$ tuples/sec, it is possible to observe that lower is the sliding window length, higher is the offered bandwidth.



(a) Service time (b) Scalability

Figure 5.7: Effects of the input stream rate on the service time (a) and scalability (b) as function of par. degree. Anticorrelated distribution. Sliding window length: 10 seconds

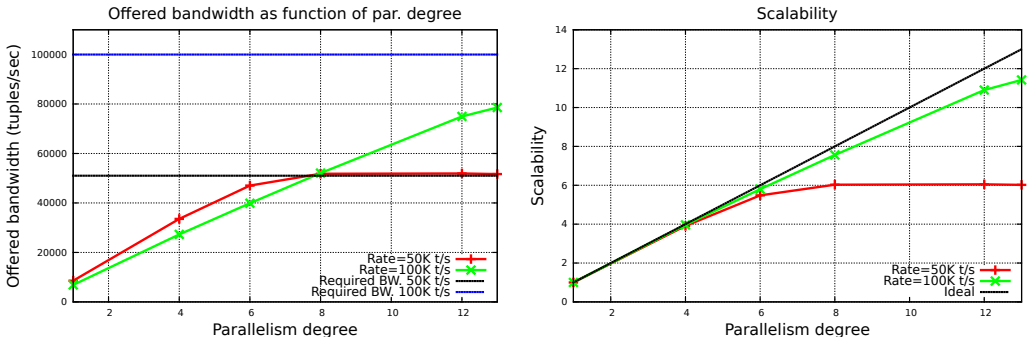
An interesting aspects can be noted comparing the Figure 5.5b with 5.7b. The former show how the sliding window length impacts on the scalability, fixed a stream arrival rate; the latter shows the effects on the scalability of varying the input stream arrival rate, fixed a window length. Both are referred to an anticorrelated distribution. In the former, with a rate of $40K$ tuples/second and a window length fixed to 30 seconds, hence with a total number of points received in a sliding window equal to $1.2M$, the bottleneck is resolved with $n = 8$. In the latter, with a rate fixed to $100K$ tuples/second and a window length of $10s$, hence with a total number of received points equals to $1M$, we are not able to resolve the bottleneck with the adopted architecture.

This behavior can be explained through the cost model discussed in Sec. 4.5. Given a stream arrival rate and a window length, the optimal parallelism degree, i.e. the one for which the module is not a bottleneck, is calculated as:

$$\bar{n} = \lambda^2 \times T_F \times W \times (1 - p)$$

it is clear that, even if the product $\lambda \times W$ is fixed[1], the contribution of the two terms to the optimal parallelism degree is different, since this last grows as the square of λ and linearly with the window length W.

[1]this implies that also the pruning probability p if fixed, since it is a function of this product.



(a) Service time (b) Scalability

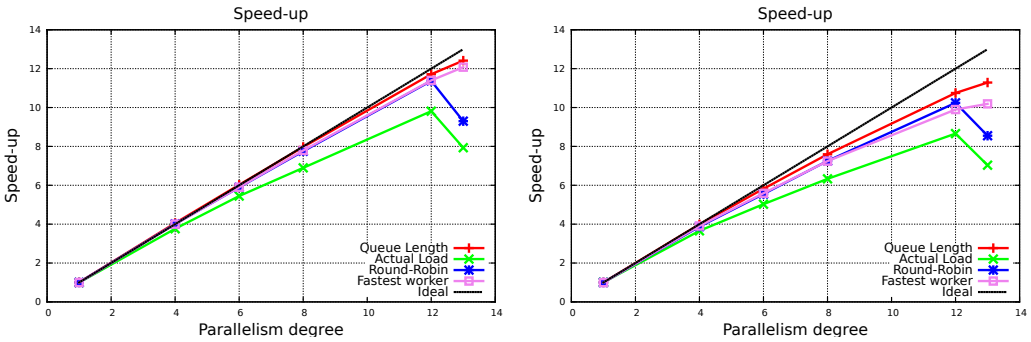
Figure 5.8: Effects of the input stream rate on the service time (a) and scalability (b) as function of par. degree. Independent distribution. Sliding window length: 10 seconds

Independent distribution Due to the higher pruning activity, there is a lower number of stored points respect to the anticorrelated one, hence the offered bandwidths in Fig 5.8a are higher than the ones of the anticorrelated case. As far as concerning the scalability, the consideration reported for the anticorrelated case are still valid.

5.3 Effects of the partitioning scheme

In this experiment the impact of the selected partitioning scheme on the computation speed-up is measured. The heuristics on which this experiment is based are the one presented in Section 4.1.2. In addition we consider also the round-robin owner selection in order to compare the load-aware heuristics with a load-unaware one. The speed-up is measured in relation to the service time of the sequential module with the same parameters and where, obviously, no partitioning is required.

Since we want to observe how the load-balancing techniques impact on the speed-up, we choose, for each data distribution, a set of parameters (λ, W) such that the module is always bottleneck.



(a) Anticorrelated (b) Independent

Figure 5.9: Effects of the partitioning scheme on the speed-up, both in anticorrelated (a) and independent (b) case. The asynchrony degree set to 10K points, except for *Fastest worker* in which is set to 1.

Anticorrelated distribution The results with this type of distribution are reported in Fig. 5.9a. It is the one with the lowest pruning activity, hence the load unbalancing is minimal. As can be observed, the *Queue length* partitioning scheme is the one with the highest speed-up. The *Actual load*, instead, is the one performing worse: as explained in Sec. 4.1.2, this heuristic does not take in account the presence of a queue between the emitter and each worker. When the asynchrony degree of these queues is high, such as in this case, this approach introduces a non-negligible error in the selection of the least-loaded owner.

Independent distribution As said, this distribution leads to an higher pruning activity, hence to more load unbalancing. In Fig. 5.9b is reported the speed-up of all the partitioning schemes with respect to this kind of data distribution. In all the cases, the higher load unbalancing leads to a lower speed-up than the one reached in the anticorrelated case. However, even with this configuration, the *Queue length* heuristic is the one taking the best counter-measures, hence the one reaching the best results.

Correlated distribution Since this distribution is characterized by an intensive pruning activity, it is the case presenting the highest load unbalancing. As shown in Fig. 5.10, the hierarchies are unchanged, except for *Fastest worker* which is the one that, in this case, have the worst performances.

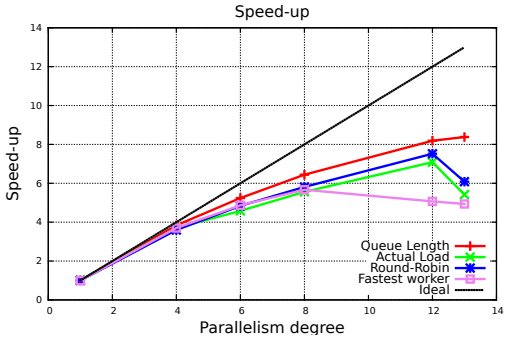


Figure 5.10: Effects of the partitioning scheme on the speed-up in the correlated case. The asynchrony degree set to 10K points, except for *Fastest worker* in which is set to 1.

Effects of SMT It is interesting to note what happens, with the anti-correlated and independent distribution, when we start to allocate multiple workers on the same core with different SMT contexts, at $n = 13$. While the speed-up of *Fastest worker* and *Queue length* continue to grow, the others are subject to a significant slow-down. For the *Fastest worker* heuristic the explanation is obvious: a point is assigned to the first worker that is able to accommodate it in its queue. If a worker is slower, as in the case of $n = 13$, the emitter will assign to it a minor number of points, hence the speed-up will continue to grow in a sub-linearly way with the parallelism degree. As far as concerning the *Queue length* partitioning scheme, it is based on both the actual load of the workers and the number of enqueued points for which each worker has been elected as the owner. A slower worker will serve slowly the points in its queue, hence it will be designed as the owner for a minor number of points. In conclusion, we can state that these two partitioning schemes aim to balance the service time of the workers, instead of their merely loads. This result is interesting because it is applicable also in cases where the T_F presents a non-negligible variance, such as in the case of the R-Tree-based implementation of the module.

The other partitioning schemes, instead, perform the owner selection in a totally independent way from the workers' service time: even if the load is balanced, the service time of the module will be dominated by the one of the slowest workers.

5.4 Maximum sustainable rate

In this experiment the input stream arrival rate that the module is able to sustain before it became bottleneck is measured, for each parallelism degree.

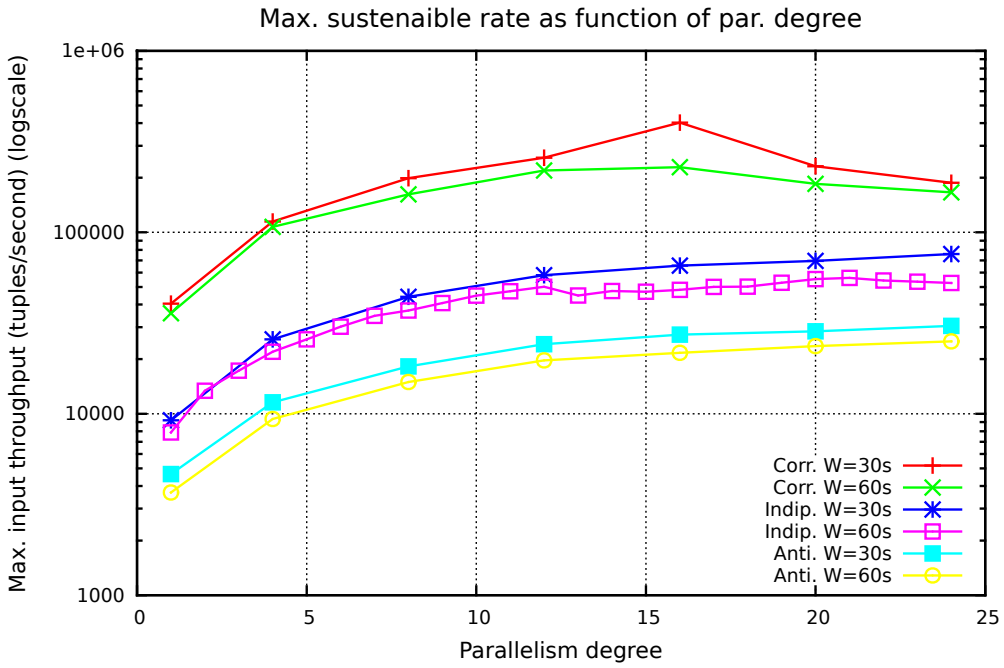


Figure 5.11: Maximum sustainable rate. Sliding window length fixed to 10 second. Logscale on y-axis.

In Fig. 5.11 is possible to note the importance of data distribution. We remark that, as discussed in Section 4.5, fixed a window length, the module is not bottleneck for:

$$\lambda \leq \sqrt{\frac{n}{W \times (1-p) \times T_F}}$$

The T_F measured on the test architecture is about 1.7×10^{-6} second. Clearly, in the hypothetical case in which no pruning operation is performed ($p = 0$), we expect that the maximum sustainable rate grows as the square root of n, regardless the specific data distribution. However, the pruning activity is present and how much pruning is performed is indicated by the pruning probability p, which: a) is different for each data distribution; b) grows as the number of received points per sliding window, i.e. the $\lambda \times W$ product. Therefore, in a way dependent on the two above factors, the maximum

sustainable rate grows faster than the square root of n. Nevertheless, you have to consider that W appears in the denominator of the sustainable bandwidth formula: higher is W, lower is the sustainable λ.

Chapter 6

Conclusion

In this thesis we proposed a parallel module which applies the skyline operator over a stream of d-dimensional points. The basic algorithm used as the starting point of the parallelization is SIT, presented by Tao & Papadias in [26]. Our parallelization pattern is a data-parallel map, followed by a reduce phase.

Since the SIT algorithm prunes the points which are made obsolete by the arriving of new ones, its load is highly variable over the time. This behavior impacts on the proposed parallelization introducing potential load unbalancing that, if not contrasted, can nullify its benefits in terms of service time. To reduce this phenomena, we propose a set of load-aware partitioning schemes, in particular it has been shown that *Queue length* heuristic is designed with the aim to minimize the load unbalancing even if the module is bottleneck and the asynchrony degree of the queues between the emitter and the workers is high.

Two versions of the parallel module have been implemented, one utilizing the R-Tree as indexing data structure for the stored points, as proposed by the authors of SIT in [26], and one based on a random-access data structure. The results show that the R-Tree-based version outperforms the vector-based one only when the number of received points per sliding window is over a certain threshold. In according to the literature, this threshold is possibly out from the applicative domains of this kind of computation. Due to this motivation and the fact that the R-Tree's performances depend on the data distribution of the stored points, the vector has been chosen as data structure for the storage of both skyline and non-skyline points. Moreover, we discussed that the R-Tree-based implementation, and in general a module whose service time can be expressed as a logarithmic function of the size of its internal state, cannot reach a linear scalability due to theoretical reasons.

For each incoming point a reduce phase is performed in order to calculate its *skyline influence time*. If the reduce is synchronous, the owner of that point must wait for the participation of all the workers to the collective operation. If the load is unbalanced, this behavior has a negative impact on the module service time. Since our target is to minimize it, we propose an optimization, called *asynchronous reduce*, that avoid the owners to wait the completion of calculation of the SITs relative to the points assigned to them.

In Section 4.5, is presented a cost model for calculating the maximum input stream arrival rate that the parallel module is able to sustain before it became bottleneck, given a parallelism degree. With this cost model we are able to show how the spatial distribution of the points impacts on the module service time.

Finally, in Chapter 5, the results of a set of experiments relative to the various aspects of the computation is presented. In this section we show that maintaining the load unbalancing to negligible levels (with the proposed partitioning schema), and using the optimized version of the reduce, we are able to reach good scalability levels.

Future works As a future work, the *Queue length* partitioning scheme can be improved in order to take in account also the pruning that each element in the workers' queue can cause. This approach will lead to a better approximation in forecasting the worker's load and hence to a better load balancing. A possible extension to our work is to make the module able to satisfy multiple skyline queries, i.e. different sliding window lengths and/or dimensions. The resulting module will have multiple output streams, one for each query. Finally, an explicit cost model for calculating the pruning probability p could be found: combining it with our cost model, it will be possible to analytically find the maximum stream arrival rate that the module is able to sustain knowing just the T_F, which is architecture dependent.

Appendices

Appendix A

MINI

This algorithm is presented in [15]. It finds the skyline over the points contained in an R-tree with the constraint that only the ones dominated by a given point p must be taken into account. It utilizes a min-heap data structure that can contains two type of objects: points or R-tree's nodes. In both cases, the Manhattan distance from the origin of the data space is associated with each object and the min-heap is based on this last one.

Algorithm 9 $Mini(N, p)$. N is root of the R-tree, p is the constraint point

```
1: miniSkyline ← ∅
2: insert (N, 0) into Heap
3: while Heap isn't empty do
4:   if Heap.top is a point then
5:     point = pop Heap
6:     if point is not dominated by anyone in miniSkyline then
7:       insert point in miniSkyline
8:     end if
9:   else
10:    node = pop Heap
11:    if node.upper_right ∈ p.DR then
12:      if node is a leaf node then
13:        local_skyline = compute the skyline over the points in node
14:        for each c ∈ local_skyline do
15:          if c ∈ p.DR then
16:            insert (c, Dist_c) into Heap
17:          end if
18:        end for
19:      else
20:        for each c in node.children do
21:          insert (c, Dist_c) into Heap
22:        end for
23:      end if
24:    end if
25:  end if
26: end while
```

Appendix B

FindDominatedPoints

Given an R-Tree and a point p, this operation finds the set of points dominated by p. It is employed in Lazy for the pruning of the points stored in DB_{sky}.

The algorithm utilizes an heap whose elements are the R-Tree nodes to analyze.

Algorithm 10 $findDominatedPoints(N, p)$. N is the R-tree's root; p is the dominating point;

```
1: insert N into Heap
2: while Heap is not empty do
3:    node = pop Heap
4:    if p dominates the upper-right corner of node then
5:       if node is a leaf then
6:          for each point x ∈ node do
7:             if p dominates x then
8:                insert x into Dom_p
9:             end if
10:         end for
11:      else
12:         for each Child c ∈ node do
13:            insert x and its distance into Heap
14:         end for
15:      end if
16:   end if
17: end while
```

Appendix C

FindSIT

This algorithm is presented in [26]. Given an R-Tree and a point p, it finds the *skyline influence time* of p. It is based to an R-Tree in which each intermediate node E stores the number $E.t_{exp}^{max}$ equals to the maximum expire time of the points in its subtree. The algorithm utilizes a max-heap H whose elements couples $< E, key >$ where E is an R-Tree node.

Algorithm 11 $findSIT(T, p)$. T is the R-Tree's root; p is the point whose the SIT has to be calculated;

```
1: p.t_sky = current time
2: for each node E in the R-Tree root T do
3:    if E intersects p.ADR and E.t_exp^max > p.t_sky then
4:       insert < E, E.t_exp^max > in H
5:    end if
6:    if E is covered by p.ADR and E.t_exp^max > p.t_sky then
7:       p.t_sky = E.t_exp^max
8:    end if
9: end for
10: while H is not empty do
11:    < E, E.t_exp^max > = pop from H
12:    if E.t_exp^max < p.t_sky then return
13:    if E points to an intermediate node N then
14:       for each entry E' in N do
15:          if E' is covered by p.ADR and E'.t_exp^max > p.t_sky then
16:             p.t_sky = E'.t_exp^max
17:          else if E' intersects p.ADR and E'.t_exp^max > p.t_sky then
18:             insert < E', E'.t_exp^max > in H
19:          end if
20:       end for
21:    else
22:       //E points to a leaf node N
23:       for each record r in N do
24:          if r dominates p and r.t_exp > p.t_sky then
25:             p.t_sky = r.t_exp
26:          end if
27:       end for
28:    end if
29: end while
```

Bibliography

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

[2] A. Vlachou, C. Doulkeridis, Y. Kotidis. 2008. *Angle-based space partitioning for efficient parallel skyline computation. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08).* ACM, New York, NY, USA, 227-238

[3] B. Cui, H. Lu, Q. Xu, L. Chen, Y. Dai, Y. Zhou. 2008. *Parallel Distributed Processing of Constrained Skyline Queries by Filtering. In Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE '08).* IEEE Computer Society, Washington, DC, USA, 546-555

[4] C. Papadimitriou, M. Yannakakis. 2001. *Multiobjective query optimization. In Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS '01).* ACM, New York, NY, USA, 52-59

[5] D. Buono, T. De Matteis, G. Mencagli, M. Vanneschi. *Towards a Methodology for Parallel Data Stream Processing: application to Parallel Stream Join. University of Pisa*, 2013

[6] D. Kossmann, F. Ramsak, S. Rost. *Shooting Stars in the Sky: An Online Algorithm for Skyline Queries, Proc. Int'l Conf. Very Large Databases (VLDB)*, pp. 275-286, 2002

[7] D. Papadias, Y. Tao, G. Fu, B. Seeger. *An Optimal and Progressive Algorithm for Skyline Queries, Proc. ACM SIGMOD*, pp. 467-478, 2003

[8] D. Papadias, Y. Tao, G. Fu, B. Seeger. *Progressive skyline computation in database systems. ACM Transactions on Database Systems (TODS)*, 30(1):41–82, 2005

[9] F. Preparata, M. Shamos. *Computational Geometry - An Introduction*. Springer-Verlag, 1985. 1st edition: ISBN 0-387-96131-3; 2nd printing, corrected and expanded, 1988: ISBN 3-540-96131-3; Russian translation, 1989: ISBN 5-03-001041-6

[10] H. T. Kung, F. Luccio, F. P. Preparata. 1975. *On Finding the Maxima of a Set of Vectors*. J. ACM 22, 4 (October 1975), 469-476

[11] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, *Skyline with Presorting*, Proc. Int'l Conf. Data Eng., pp. 717-719, 2003

[12] K.-L. Tan, P.-K. Eng, B. C. Ooi. 2001. *Efficient Progressive Skyline Computation*. In Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)

[13] L. Chen, K. Hwang, J. Wu. 2012. *MapReduce Skyline Query Processing with a New Angular Partitioning Approach*. In Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW '12)

[14] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, M. Torquati, *An efficient unbounded lock-free queue for multi-core systems*, in Proc. of 18th Intl. Euro-Par 2012 Parallel Processing, vol. 7484, Rhodes Island, Greece, aug 2012, pp. 662–673

[15] M. Morse, J. M. Patel, and W. I. Grosky. 2007. *Efficient continuous skyline computation*. Inf. Sci. 177, 17 (September 2007), 3411-3437

[16] M. Vanneschi, 2009. *Architettura degli elaboratori*. Pisa University Press

[17] M. Vanneschi, 2014. *HPC course notes*. University of Pisa

[18] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, *The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles*, Proc. ACM SIGMOD, pp. 322-331 1990

[19] N. Roussopoulos, S. Kelley, F. Vincent. 1995. *Nearest neighbor queries*. SIGMOD Rec. 24, 2 (May 1995), 71-79

[20] P. Wu, C. Zhang, Y Feng, B. Y Zhao, D. Agrawal, A. E. Abbadi, *Parallelizing skyline queries for scalable distribution*. in Proc. EDBT, 2006, pp. 112-130

[21] S. Borzsonyi, D. Kossmann, K. Stocker. *The skyline operator*. ICDE, 2001

[22] S. Park, T. Kim, J. Park, J. Kim, H. Im. *Parallel Skyline Computation on Multicore Architectures*, Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on , vol., no., pp.760,771, March 29 2009-April 2 2009

[23] S. Wang; B.-C. Ooi; A. Tung, L. Xu. *Efficient Skyline Query Processing on Peer-to-Peer Networks*, Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on , vol., no., pp.1126,1135, 15-20 April 2007

[24] V. Hristidis, N. Koudas, Y. Papakonstantinou. 2001. *PREFER: a system for the efficient execution of multi-parametric ranked queries*. In Proceedings of the 2001 ACM SIGMOD international conference on Management of data (SIGMOD '01), T. Sellis and S. Mehrotra (Eds.). ACM, New York, NY, USA, 259-270

[25] X. Lin, Y. Yuan, W. Wang, Hongjun Lu. 2005. *Stabbing the Sky: Efficient Skyline Computation over Sliding Windows*. In Proceedings of the 21st International Conference on Data Engineering (ICDE '05). IEEE Computer Society, Washington, DC, USA, 502-513

[26] Y. Tao, D. Papadias, *Maintaining Sliding Window Skylines on Data Streams*, IEEE Transactions on Knowledge and Data Engineering, vol. 18, no. 3, pp. 377-391, March 2006

[27] Y. Theodoridis and T.K. Sellis, *A Model for the Prediction of R-Tree Performance*, Proc. ACM Symp. Principles of Database Systems, pp. 161-171, 1996

[28] Y. W. Lee, K. Y. Lee, M. H. Kim. *Efficient computation of multiple sliding window skylines on data streams*. Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference on , vol., no., pp.951,956, Nov. 30 2010-Dec. 2 2010