Exploiting multi-core parallelism with pipelining to solve skyline queries

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 94

Systems Group, Department of Computer Science, ETH Zurich

Exploiting multi-core parallelism with pipelining to solve skyline queries

by

Patrick Bänziger

Supervised by

Louis Woods
Prof. Gustavo Alonso

September 6, 2013

inf | Informatik
Computer Science

Abstract

Modern computer processors provide an increasing amount of parallel computation resources. Harnessing the full potential of these resources requires approaches that are adapted to this parallelism, especially as the core count increases.
Solving skyline queries is a well-known, computationally intensive problem in the context of database systems. Many solutions and optimizations have been suggested, most of which optimized parts of the implementation or used pre-computed data structures to speed up the computation. However, the majority of approaches did not harness the potential of modern multi-core processor architectures. The only approach that exploits multi-core parallelism so far, is based on an input-partitioning scheme, which is not optimal as we will show in this thesis.

Pipelining, or pipeline parallelism, is a concept which allows to increase the throughput when processing data elements, but at the cost of a higher latency for individual elements. In this thesis, we present how a pipelining approach can be used to solve skyline queries on multi-core systems and compare it against a state-of-the-art implementation based on input-partitioning. We use a doubly-linked list of computation nodes that shift and transfer data elements between each other to calculate the result. Our experimental results show that our approach outperforms the state-of-the-art implementation by an order of magnitude in all relevant workloads.

We develop a template which allows to implement pipelined approaches for related problems with only a small development effort and apply the template to skyline computation and sorting. Furthermore, we describe the properties which a task needs to exhibit to make use of our template.

Author's Note: This document is optimized for electronic viewing with electronic hyperlinks, both within the document and to external resources.

Acknowledgements

The author would like to thank Louis Woods of the Systems Group at ETH Zurich for his invaluable advice and supervision of this thesis, and Zaheer Chothia for his shared insights and knowledge of hidden compiler features and analysis tools.

Contents

1 Introduction

1.1 Motivation

Modern processor designs have departed from the previous approach of a single unit running at high clock speeds and now provide multiple cores which provide parallel compute resources. To properly utilize these resources, we need to develop parallelized solutions to existing and new problems.

Determining the skyline of a set of points (see section 3) is such a compute-intensive problem which has efficient and well understood single-threaded implementations.

Parallel programming is a still an active area of research and tries to harness the parallel computation resources provided by modern computer systems. An extreme case are FPGA systems: Field Programmable Gate Arrays (see subsection A.1. These offer unmatched parallelism using configurable hardware circuits, but their design and limitations often require new approaches to solve a problem.

One such approach was presented by Woods et al. [14]: An implementation to solve skyline queries on FPGA system which performed very well compared to the state-of-the-art implementation on powerful servers.

1.2 Goal

The aim of this thesis is to explore how to adapt the approach taken with the implementation by Woods et al. [14] to multi-core systems. We aim to evaluate the viability of the ported approach and measure and analyse the performance and compare it to state-of-the-art implementations.

Further, we aim to identify related problems which can be solved by such an approach, develop a template implementation and to evaluate how high the effort is to adapt the approach to these problems.

2 Related work

Kung et al. [1] presented a mathematical definition of finding the maxima of a set of vectors in 1975. In 2001, Börzsönyi et al. [2] presented the problem in a database setting as "solving skyline queries" and presented two possible approaches to implement it (block nested loops, divide and conquer). A first "online" implementation which produced result tuples continuously (not only at the end of the algorithm) was presented by Kossmann et al. [3]

The presentation of the problem was followed by research on various optimizations of the approaches: Chomicki et al. [4] used sorting as a first phase before determining the skyline in the second phase. This approach allowed to immediately output tuples as skyline tuples when they were added to the window in the second block nested loops-like phase. Branch-and-Bound Skyline was proposed by Papadias et al. [5] and operated on an R-tree index structure.
Lee et al. developed an index structure based on the Z-Order [6] and used knowledge about the resulting partitioning to discard entire regions from consideration for the skyline. This approach showed a high performance, however was not applicable if the index structure couldn't be precomputed, e.g. if data were made available as a stream. A comprehensive overview of the

sequential approaches and their complexities was presented by Godfrey et al. [7]

Most research had focused on traditional optimizations and did not take advantage of the paradigm change in computer architecture: The shift from single- to multi-core processors.
One of the first implementations to leverage super-scalar features of modern processors was Cho et al. [8] Their implementation used vectorization to speed up dominance tests in SIMD architectures by parallelizing the comparisons of the respective dimension elements. And as the dominance test is likely the most executed function in computing the skyline, the performance of this function is vital to the overall performance of an approach.

With the advent of grid-computing, more research was done on how to handle skyline queries on distributed data or how to leverage the power distributed systems to process such queries. Cui et al. [9] proposed a parallel distributed skyline for data that is located at different sites. The approach partitions sites into incomparable groups which can execute the skyline query in parallel and transfer some specially chosen tuples between the sites to eliminate false positives before transmitting the intermediate results for merging. Such early elimination reduces the size of intermediate results and thus transmission and computation time.
Vlachou et al. [10] proposed a novel angle-based partitioning of the data space using hyperspherical coordinates, This approach for distributed processing allows to discard more points per partition than other partitioning approaches, resulting in smaller intermediate results. Köhler et al. [11] proposed a similar approach but used hyperplane projections as a computationally cheaper partitioning approach.

Still, many approaches did not take into account that modern processors now featured multiple cores. This changed with Park et al. [12]: They presented a state-of-the-art parallel algorithm for processing skyline queries on multi-core processors. It is based on a input-partitioning approach which maps the (in-place) skyline function to partitions of the input set and then merges the results. It assumes that the input fits into main memory.

Teubner et al. [13] presented a parallel pipelined approach for multi-core systems to solve joins in a database setting. A pipelined approach was not yet used in research solving skyline queries, until Woods et al. presented an pipelined FPGA-based implementation [14].
FPGA systems offer massively parallel computation capabilities, however require significant knowledge and handling of low-level hardware details, compared to a commodity computer.

The focus of this thesis is solving skyline queries on multi-core hardware where all data is either located at or streamed to one device and no pre-processing can be done. With Park et al. there is only one main parallel approach for such hardware. Our aim with this thesis is to see if the pipelined approach taken by Woods et al. can be adapted and if it shows good performance on multi-core hardware.

3 The skyline operator

3.1 Introduction

The skyline operator was presented initially by Börzsönyi et al. [2] A skyline query over a multi-dimensional set of data points will return a subset, such that in this subset no point *dominates* any other point. Informally, this is often illustrated by the following two-dimensional example.

Example When searching for a hotel, two dimensions to consider are the price of the hotel and the distance to the beach. For both dimensions, smaller values are preferable. Hotels which are close to the beach often are more expensive than hotels which are distant. If we do not have a scoring function that weighs the distance and price, we cannot establish a 'better than'-order. We can only eliminate hotels from the candidates if there is another hotel that is better in at least one dimension and equal in the remaining dimensions. E.g.: A hotel can be removed from consideration if there is a cheaper hotel with the same (or smaller) distance to the beach.

The remaining hotels form the skyline of the hotel set, which can give the user/customer an overview and a better understanding of the trade-offs that need to be made when choosing a hotel. Figure 1 illustrates this, the diamonds being the hotels in the skyline set.

Figure 1: Two-dimensional skyline example

3.2 Formal definitions

Given a d-dimensional space $T = \{d_0, d_1, \ldots, d_{n-1}\}$ and a set of points $M = \{p_1, p_2, \ldots, p_n\}$ where each point p_i is a point in the space T, we can define the problem as follows.[1]

Definition 1. Dominance
Given two points $p, q \in T$: We say q *dominates* p (notation: $q \succ p$) if and only if:

$$\forall i \in T : q_i \geq p_i \wedge \exists i \in T : q_i > p_i$$

Note that the $>$ / $\geq$ relation chosen to compare the elements of dimensions could also be replaced by $<$ / $\leq$, depending on the problem statement.

[1] These definitions were introduced with a slightly different notation by Lee et al. [6]

Definition 2. Skyline
A point $p \in M$ belongs to the skyline of M (S_M), if and only if it is not dominated by any other point:

$$\nexists q \in M - \{p\} : q \succ p \quad \leftrightarrow \quad p \in S_M$$

Property 1. Incomparability
Two points $p, q \in T$ are incomparable if and only if neither dominates the other:

$$p \not\succ q \wedge q \not\succ p \quad \leftrightarrow \quad p \diamond q$$

Note that this implies that this implies that a point does not dominate itself.

Property 2. Transitivity
The dominance property is transitive:

$$p \succ q \wedge q \succ r \quad \Longrightarrow \quad p \succ r$$

This transitivity property is very significant for any implementation which calculates the skyline: As soon as a point q is dominated by a point p, it can be discarded. This because all points that would be dominated by q will be dominated by p as well.

Property 3. Asymmetry
The dominance property is asymmetric:

$$p \succ q \quad \Longrightarrow \quad q \not\succ p$$

Property 4. Reflexivity
The dominance-relation is *irreflexive*, i.e. a point cannot dominate itself. The incomparability-relation however is *reflexive*.

$$p \not\succ p$$

$$p \diamond p$$

Further, as noted in [2], the skyline has the following property:

Property 5. Score maximization
For any monotonic (strictly increasing) scoring function $f : T \to \mathbb{R}$ and any set of points M, the skyline of M will always contain the point $p \in M$ which maximizes the scoring function.

$$p = \underset{t \in M}{argmax}\,(f(t)) \quad \Rightarrow \quad p \in S_M$$

Proof. Without loss of generality, let us assume that we use the $>$-relation to compare dimension elements, as used in definition 1.
We prove the property by contradiction: Assume that a point p maximizes the monotone scoring function f but is not part of the skyline. Then there had to exist a point q which dominates p. And by definition 1 this implies that all elements of q are greater or equal to the corresponding elements of p and at least one must be strictly greater. For any monotonic (strictly increasing) scoring function, this would however imply that $q > p$, which contradicts the assumption that p maximizes f. □

The skyline problem is an instance of the maximum-vector problem [7].

3.3 Algorithms

This section enumerates some basic concepts for determining the skyline which do not use any preprocessed data structures.

3.3.1 Naive

The naive unparallelized way to evaluate a skyline query over a set of data points would be to iterate over the set and check every point against every other in the set, removing dominated points and output the remaining ones as skyline points.

Algorithm 1 Naive computation of the skyline

```
1:  procedure naiveSkyline(set M)
2:      for i=0 ... M.size-1 do
3:          if ! M.get(i).valid then
4:              continue
5:          end if
6:          dominated = false
7:          for j=i+1...M.size-1 do
8:              if ! M.get(j).valid then
9:                  continue
10:             end if
11:             if M.get(i) ≻ M.get(j) then
12:                 M.get(j).valid = false
13:             else if M.get(j) ≻ M.get(i) then
14:                 dominated = true
15:                 break
16:             end if
17:         end for
18:         if ! dominated then
19:             output(M.get(i))
20:         end if
21:     end for
22: end procedure
```

3.3.2 Divide and conquer

This approach divides the input set into partitions, determines the skyline set in the partitions and then merges the skylines of the partitions. It can be parallelized to perform the skyline computation on the partitions in parallel, and to perform several merge operations in parallel.

Such an approach performs very well, if the skyline is distributed evenly (position-wise) within the dataset and the skyline is small: The results of the partitions will be small, resp. contain a small number of "false positives" (points that will not be part of the final skyline) and the merge operations correspondingly cheap to perform.
The case in which the approach might not perform well (depending on the exact merging approach) is if the intermediate results are very large, e.g. if the partitions result sizes are very unbalanced. This would lead to the a larger number of false positives in the intermediate results and fast merge operations.

3.3.3 Block nested loop

An issue of the naive algorithm is the extremely poor cache utilization: Assuming that the input size far exceeds the cache size, only the candidate point can be kept in the cache in an iteration of the outer loop. The other points which the naive algorithm iterates over will not be cached!

Instead of checking a data point against every other data point in the set for dominance, the block nested loop join uses a small subset ("block") to test against the set. As the elements of the block are frequently used, more elements are likely to be cached. After the block has been compared to the whole data set, the next block can be tested. It is important to test a block against the whole data set to make sure that the elements of a block are compared against each other, too! This approach can be seen in algorithm 2.

Algorithm 2 Bock nested loop computation of the skyline

```
1:  procedure BLOCKSKYLINE(set M)
2:      while block = nextBlock(M) do
3:          for j in M do
4:              if ! j.valid then
5:                  continue
6:              end if
7:              for b in block do
8:                  dominated = false
9:                  if b ≻ j then
10:                     j.valid = false
11:                 else if j ≻ b then
12:                     dominated = true
13:                     break
14:                 end if
15:             end for
16:             if ! dominated then
17:                 output(b)
18:             end if
19:         end for
20:     end while
21: end procedure
```

This approach could be parallelized by processing the possible blocks in parallel.

4 Shifter List for skyline queries

4.1 Motivation

In [14], Woods et al. presented an approach to solve skyline queries (see section 3) on FPGAs. Their approach described a hybrid of algorithm and data structure and used a set of compute nodes which transferred their data to their adjacent nodes. Effectively, the resulting data structure presented was similar to a doubly-linked list.

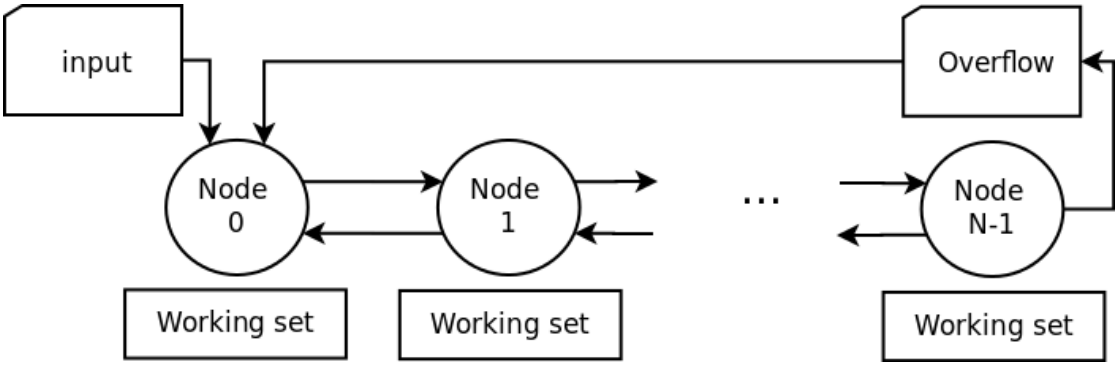


Figure 2: Overview of Shifter List architecture

Such a "Shifter List" has several desirable properties: Each node operates on a local data set with a block-nested-loop (BNL) approach. Such local data access would indicate good spatial locality which typically is advantageous for cache and overall performance. Local data access is also a requirement for good performance systems with NUMA characteristics (see subsection A.2) as these systems incur performance penalties when accessing remote memory compared to local memory.
Further it communicates only with adjacent nodes which makes the communication pattern significantly simpler and leaves room for possible optimizations.

Because of these potential benefits, we wish to explore the possibility of implementing such a Shifter List on a modern multi-core computer.

Terminology We will use the term "Shifter List" throughout this thesis to refer to the structure presented in this section.

4.2 Introduction

This section describes the general concepts and behaviour of a Shifter List implementation which solves skyline queries. In section 5, we will describe the actual implementation details.

A Shifter List uses the concept of nodes: A **node** is a (virtual) compute unit with associated memory. Nodes are connected to their direct neighbours, forming a doubly linked list as seen in Figure 2.
Nodes can communicate only to their neighbours to forward data and transfer potential results. These limited communication connections are vital to the FGPA implementation, as it reduces the required amount of interconnection wires and routing compared to a graph of fully connected nodes.

Data points are "streamed though" the nodes, enabling *pipeline parallelism*. Every node only operates on its local dataset from associated memory and the data elements it receives from its neighbours.
This section describes the generalized implementation necessary to solve skyline queries.

4.2.1 Terminology

The **data stream** is the flow of data points from node 0 to node $n-1$. We name this direction "down" and the reverse direction "up". Thus nodes further "upstream" have smaller a smaller numeric id than nodes further "downstream". We will use the term "working set" to refer the local memory maintained by each node.

4.3 Local candidate set

Every node keeps a local set of candidates that could be in the skyline in his working set. We call the set of points maintained within the working set the "candidate set". The original Shifter List implementation for FPGAs has a set size of one, i.e. only one single element per node that is a skyline candidate. On a multi-core implementation we have more memory available, so this restriction to a single element is unnecessary.

4.4 Overflow

Naively, we could have large enough working sets to store the whole skyline as candidates. However even if enough memory were available to store the skyline, very large working sets are not desirable: Due to their size, their spatial and temporal locality would be worse, as every data point needs to be compared to the entire candidate set at each node if incomparable. If the size exceeded the cache size of a processor (core), this would lead to increased cache misses which would in turn lead to a lower performance for the node.

Instead, we use an overflow data structure. The implementation of this is arbitrary - the only requirement is that it supports the FIFO order and could store the whole input[2].

Now, if a point has reaches the last node and cannot be inserted into the local working set, it has not been compared to all points yet and needs to begin again. It is inserted into the overflow and, once all elements from the input have been read, will be re-read from the overflow by the first node.

4.5 Handling end of input

When the input has been read completely by the first node, the first node treats the overflow as new input and the last node writes to a new overflow.
This can be achieved for example by swapping the input and overflow (and taking care of 'flushing' the data stream before swapping) or by using a wrap-around queue from the last to the first node.

Once the first node does not find any data points any more in the input after such a swap, the algorithm can terminate. By property 6, all elements in the working sets can be output.

4.5.1 Order preservation

By reading all input first before reading from the overflow - and by always processing the data points in FIFO (first in, first out) order - we ensure that data points in the data stream flowing downwards are never reordered. This is a necessary condition for this implementation because if reordering were possible, reordered points might not be compared to all input, leading to false positives in the skyline.

[2]Practically, it will never need to store the whole input, but in the most extreme case $\#input - window_{n-1}$

4.6 Accepting into the candidate set

To ensure that all points are compared to each other, we only accept data points into the candidate set at the last node, if the candidate set has still space. This way we can be sure that when a point is inserted into the candidate set of the last node, it has been compared against all candidates at every node.

If we accepted it at a node n_i further upstream which had a non-empty candidate set, the data point would have not been compared to the local candidate sets of the nodes $n_{i+1}, \ldots, n_{n-1}$ and could be output without being compared to all data points in the input set, which would violate definition 2.

4.7 Output condition

To guarantee that a point x is in the skyline, we have to ensure that no other point exists that dominates x (by definition 2). Thus, every point that is output as part of the result must have been compared against all other points.
We can safely exclude the already eliminated points by property 2. This means that a point in the candidate set can be output if and only if it has been compared to all remaining points.

Now we need to determine when a point has been compared against all remaining other points. To do so, we use a strictly increasing global timestamp counter, such that two events cannot receive the same timestamp. All points are initialized with an empty timestamp (which always evaluates to be smaller than any other timestamp) and the first timestamp that our global counter can provide is 1.
On two occasions, we mark data points with a timestamp:

- When we insert them into a local candidate set
- When we insert them into the overflow

Property 6. A point c in the local candidate set of a node can be output if:

- It is compared against a point q in the input and it holds that: $c.time < q.time$, or
- There is no input any more to process at any node or in the overflow.

If a point c in the candidate set is compared against a point p and $p.time > c.time$, then c was inserted into the candidate set before p was inserted into the overflow. Because the insertion of the points happens only at the last node as defined previously, this implies that p must have already been compared against c (due to the order preservation property described before). Because the timestamp is strictly increasing, no data point can appear in the data stream after p with a lower timestamp. This guarantees that we can output c as part of the skyline.

4.8 Operations

4.8.1 Basic operation

The basic operation loop of a shifter list node can be seen in algorithm 3. It mainly consists of a loop that alternates between processing normal input (data points) and processing transfers. For each data point it calls a processing function which indicates by its return value what action should be performed for the data point: To drop it, forward it or to terminate the execution of

Algorithm 3 Basic operation loop of a Shifter List node

```
1: while true do
2:     if hasNormalInput() then
3:         x = getInput()
4:         if x == PT_TERMINATE then
5:             forward(x)
6:             outputAll()
7:             return
8:         end if
9:         action = process(x)
10:        if action == FORWARD then
11:            forward(x)
12:        else if action == DROP then
13:            continue
14:        end if
15:    end if
16:    processTransfers()
17: end while
```

the algorithm.

The operations used will be explained in detail in the following subsections.

4.8.2 Reading input

Because overflow and input file are swapped when necessary (subsection 4.5), the first node only has to take care of reading from the input and counting how many items it read since the last swap. If no elements are found between two swaps, then the algorithm can terminate. In algorithm 4 we see the pseudocode describing this, without specifying a specific implementation for the input/overflow swap.

4.8.3 Data forwarding

Data elements that are neither added to the local candidate set nor eliminated are forwarded to the next node. If there is no next node, the data element is inserted into the overflow.

4.8.4 Processing

This subsection describes the processing of a single data point, shown in algorithm 6.
When processing a data point on a node, it is compared against all elements of the local candidate set. First, the timestamps are compared and the candidate output if the condition specified in subsection 4.7 is fulfilled. Second, we check the actual data: If the data point is dominated by any element of the candidate set, it is immediately discarded. If it dominates a point in the candidate set, that candidate is discarded unless it is involved in a transfer (in which case it is only marked as invalidated and later deleted by the transfer algorithm). If a candidate dominates the point, the data point is discarded immediately and the processing function returns.
When the data point has been compared against all candidates of the current node, we check if we can insert it into the local candidate set. If this is the case, we drop it from the input stream, otherwise forward it to the next node.

Algorithm 4 Handling Input

```
1: procedure getInput(node n)
2:     if n.isFirst() then                          ▷ First node reads from input
3:         if wasSwapped then
4:             readCounter = 0
5:             wasSwapped = false
6:         end if
7:         i = input.read()
8:         if i == END_OF_INPUT then                ▷ All input processed
9:             if readCounter==0 then
10:                return PT_TERMINATE
11:            else
12:                performSwap()
13:                wasSwapped = true
14:                return PT_NOP                    ▷ A no-op point, will be ignored
15:            end if
16:        end if
17:        readCounter = readCounter + 1
18:        return i
19:    else
20:        return n.queueUpStreamIn.get()           ▷ Other nodes read from queue
21:    end if
22: end procedure
```

Algorithm 5 Forwarding

```
1: procedure forward(datapoint pt, node n)
2:     if n.isLast() then
3:         pt.timestamp = getNextTimestamp()
4:         overflow.insert(pt)
5:     else
6:         n.queueDownstreamOut.put(pt)
7:     end if
8: end procedure
```

Algorithm 6 Processing of a datapoint

```
1: procedure PROCESS(datapoint pt, node n)
2:     if pt == PT_NOP
3:         return DROP then                          ▷ Can discard this immediately
4:     end if
5:     for datapoint c in n.candidates do
6:         if !c.valid then
7:             continue
8:         end if
9:         if pt.timestamp > c.timestamp then
10:            if c.valid and !(c.beingSent or c.beingReceived) then
11:                output(c)
12:                dropFromCandidates(c)
13:                continue
14:            end if
15:        end if
16:        dominance = dominanceTest(c,pt)
17:        if dominance == POINT then
18:            c.valid = false
19:            if !(c.beingSent or c.beingReceived) then
20:                dropFromCandidates(c)
21:            end if
22:            continue
23:        else if dominance == WINDOW then
24:            return DROP
25:    end if
26: end for                                  ▷ The input point was not dominated.
27: insertionPos = couldInsertToCandidates(pt,n)
28: if insertionPos != -1 then                              ▷ Insertion succeeded
29:     n.candidates[insertionPos].timestamp = getNextTimestamp()
30:     return DROP
31: else
32:     return FORWARD
33: end if
34: end procedure
```

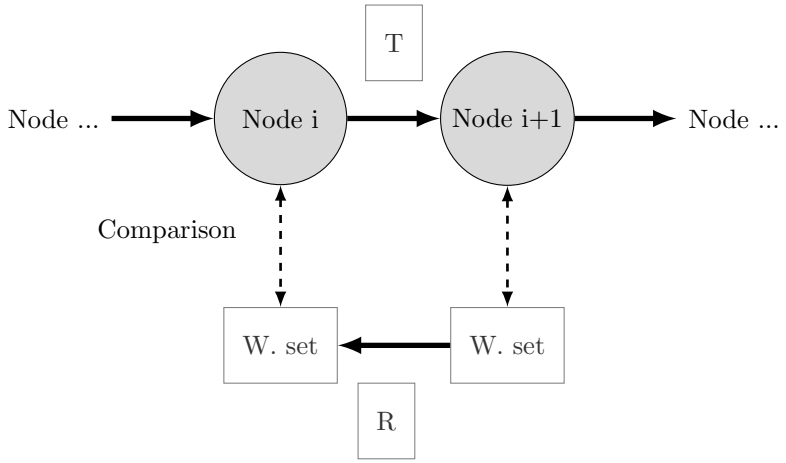


Figure 3: Transfer anomaly: Tuples T and R are not compared

4.8.5 Accepting into the candidate set

This operation was described in subsection 4.6 and is implemented in algorithm 7.

Algorithm 7 Accepting elements into the candidate set

```
1: procedure couldInsertIntoCandidates(datapoint pt, node n)
2:     if !n.isLast() or n.candidates.full() then          ▷ Cannot insert
3:         return -1
4:     else
5:         pos = n.candidates.put(pt)
6:         return pos
7:     end if
8: end procedure
```

4.8.6 Transfer

Without a transfer and because we only insert into the candidate set at the last node, the candidate set of the last node would eventually be full while all other nodes would have empty sets. Unless candidates are eliminated, the node can't accept elements into the candidate set any more until the next cycle begins and the condition for the output (subsection 4.7) is fulfilled. This would effectively reduce the implementation to a serial skyline computation on the last node.

This is the motivation for transferring (parts of) candidate sets to nodes further up the stream.

The protocol for the transfer needs to satisfy the requirement for the skyline: All data points in the input stream must be compared against any element in the candidate sets. The protocol must prevent anomalies that would violate our skyline definitions: Points which are transferred between nodes' candidate sets may neither 'miss' any points points in the data stream (see Figure 3), nor may they be output on more than one node.

4.8.7 Output at termination

When the algorithm terminates, the data points in the candidate sets are skyline points and need to be output. They may however be still in transfer between two nodes. The exact implementation depends on the transfer protocol which will be detailed in the following section.

5 Skyline: Implementation with Shifter Lists

5.1 General design

This section explains the implementation details of our Skyline implementation. The implementation uses POSIX threads[3] to spawn and manage its threads.

5.1.1 Input and output

As in the FPGA implementation, only the first node reads input from the data source. It then processes and forwards the data elements to the next node along the pipeline. If all input has been processed, it additionally forwards an "End-of-input" data element. It then processes only transfers until it receives a special message from the last node that the input and overflow have been swapped. It can then resume its computation.
All input that is neither consumed nor dropped by the last node is written to an overflow memory area.

The output memory area is shared by the nodes and synchronized over a global position indicator. This indicator is accessed only by atomic fetch&add operations to obtain a new output position for a node. This implies that the output ordering is arbitrary, as it depends on the order of transfers, data point forwarding etc.
This is different from the FPGA implementation where only the first node was able to output skyline elements. On a multi-core machine however, we can easily omit this restriction for an additional benefit: If data points don't have to be transferred to the first node for output, their slots in the local working sets become available much quicker, improving overall performance. The overhead for the small synchronization for obtaining the output position was evaluated to be not significant in this implementation.
Note that this is the only lock in our implementation after setting up the algorithm. All other operations are asynchronous.

5.1.2 Communication

The communication between two nodes is achieved by unidirectional channels. To allow for bidirectional communication, each node is assigned 2 channels per neighbour. These channels are implemented as ring buffers with a fixed capacity.
While in the FPGA implementation of Shifter List one data point at a time is transferred between the nodes, we chose to send data points in batches: Messages are implemented as structs with a fixed capacity for data points - they contain a fixed maximum number of data points and are not adjusted in size if the message is not full. They are copied to the queues when sent, and copied out of the queues when retrieved (no pointers are used).
Because we have unidirectional queues, one reader and one writer position as well as an 'occupied' flag per slot in the ring buffer are sufficient to implement the queue.

[3] `http://linux.die.net/man/7/pthreads`, retrieved on 2013-06-26

5.1.3 Transfer of working set

To transfer the working set from one core to the next we require an asynchronous transfer protocol. The protocol needs to be asynchronous to prevent that two threads need to wait to be in the proper state simultaneously to initiate the transfer (depending on the size of messages and the dimensionality of the data set, this wait time could be significant).

Further, our protocol needs to make sure that:

- Every data point in the input stream gets compared (at least once) to every element which is being transferred.
- No distinct data point can be output on two nodes.

The communication for transfers happens on the same communication channels which are used for the transfer of data points. This allows us to get an important property for free: The information which data points were by a node before a transfer was in effect.

The following protocol was adapted from the FPGA implementation:

Transfer protocol

1. The receiving node sends a HAS_SPACE message once it can accept transfers, and passes the number of free spaces of its local candidate set along.
2. The sending node receives the HAS_SPACE message.
3. The sending node sends a TRANSFER message, with the data points to transfer contained in the message (at most as many as specified in the HAS_SPACE message). It marks all data points as "being sent" in its local working set.
4. The receiving node receives the TRANSFER message. It copies all data points to its local candidate set and marks them as "being received".Then it sends an ACK message back.
5. The sending node receives the ACK and checks if any data points in its data set exist that have the "being sent" flag set and have been dominated in the meantime.
   a) If there are dominated points in its candidate set, it sends an INVALID message, with all the ids of the invalidate data points.
   b) Otherwise it sends a VALID message
6. The sending node removes all data points that had the flag "being sent" from its candidate set.
7. The receiving node receives the INVALID/VALID message. It removes the "being received" flag from any element in its local candidate set. In the case of an INVALID message, it drops any data points from the candidate set that have their id mentioned in the message.
8. The receiving node can drop any candidate from its set that was involved in the transfer and was invalidated between steps 4 and 7.

This protocol assumes that:

1. A data point cannot be transferred (again) if it is marked as "being sent" or "being received".
2. After a HAS_SPACE message has been sent from a node i, the node cannot send a HAS_SPACE message until after a transfer between node i and $i+1$.
3. Data points that are dominated but are involved in the transfer are not deleted directly, but marked as dominated/invalid.
4. A data point cannot be output if it is currently involved in a transfer.
5. When protocol messages are processed by a node, it performs any subsequent protocol step immediately without interleaving other data point processing.

The third point is required by the invalidation part of the transfer protocol: Points that are sent and dominated at the sending node need to be passed along to in the INVALID message. These points are cleaned up later by step 8 of the algorithm.

When a node can accept a transfer is configurable: In our implementation we specified a 'free' percentage as threshold for the node and note whether we have already sent a HAS_SPACE message to satisfy the second requirement. The implementation can be seen in algorithm 9.
The number of the free elements in the working set should be maintained as a counter to avoid expensive re-calculation. This calculation would be needed at every iteration of a Shifter List node, because elements could be dropped after processing input or finishing transfers.

This protocol also easily allows us to define what happens to data points in transfer when the algorithm terminates: Because the transfer acknowledgements and message for termination are sent on the same channels, we can safely assume that after a node receives a termination message, no transfers will be completed. These data points should be output by the sender.

This protocol is illustrated in Figure 4a and Figure 4b.

Correctness of Transfer The above transport protocol satisfies our requirements as follows.
Two points in the input are compared for dominance against each other exactly once, even if in transfer (provided neither is eliminated in the meantime). The following three constellations are possible:

Case a) *The point in transfer is processed after the streamed point.* Thus the streamed point is compared against the working set of the receiving node first which does not yet contain the transferred point. Therefore, the streamed point will also arrive before the ACK for the transfer at the sending node (no-reordering property) and be compared there against the copy of the point in transfer.

Case b) *The point in transfer is processed before the streamed point.* This means that the copy of the point in transfer will already be in the local working set before the streamed point is compared. Also, the ACK has already been sent (protocol steps are executed immediately). The streamed data point will be compared to the transferred point at the receiving node of the transfer. When the streamed point arrives at the sending node of the transfer, the copy of the point in transfer has already been removed because the ACK was received first (immediate execution, no reordering).

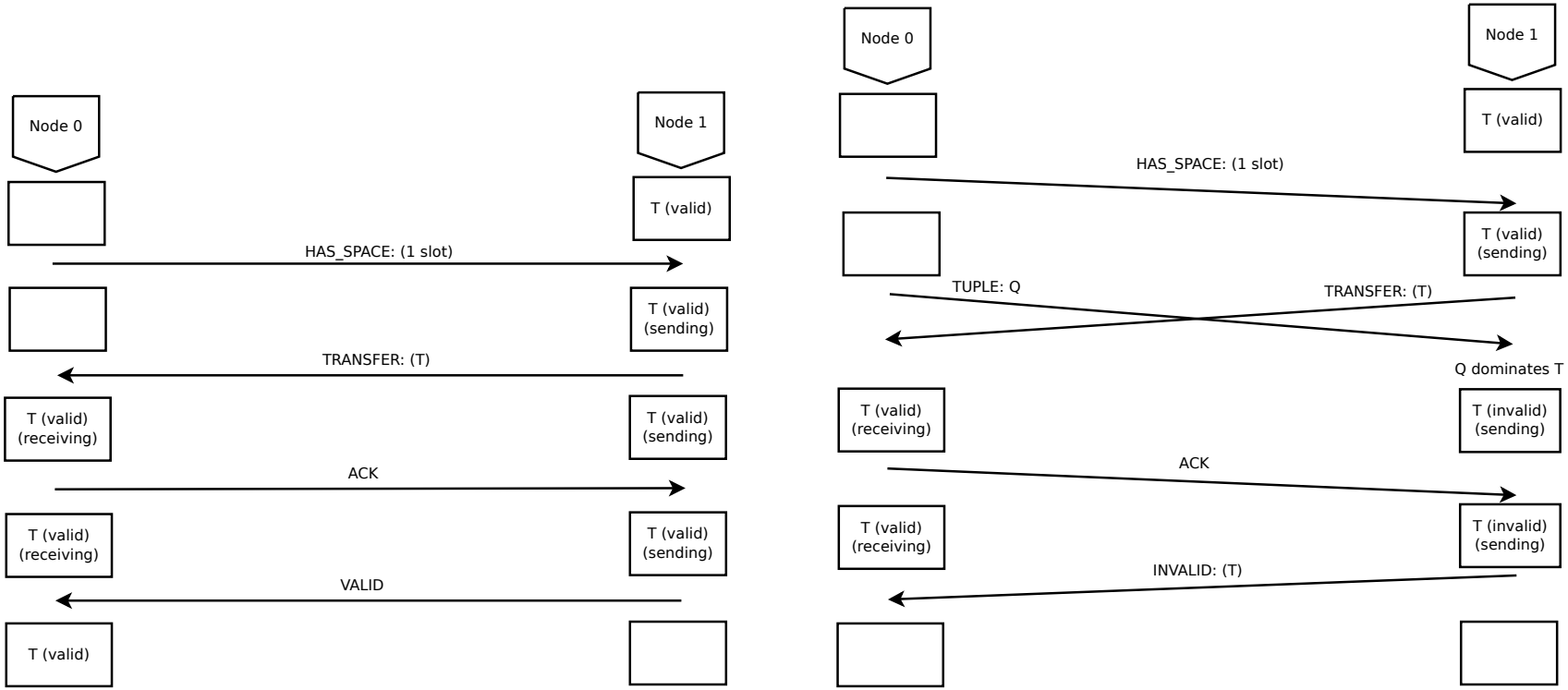
Algorithm 8 Transfer processing, Part 1

```
1: procedure processTransfers(node n)
2:     message m = n.queueUpstreamIn.get()
3:     if m.type == HAS_SPACE then
4:         max = m.get()                          ▷ Sender specified how many it can accept
5:         message t
6:         t.type = TRANSFER
7:         i=0
8:         for datapoint c in n.candidates do
9:             if i==max or i==t.msgSizeMax then
10:                break
11:            end if
12:            if !c.beingSent and !c.beingReceived then
13:                c.beingSent = true
14:                t.put(c)
15:                i = i+1
16:            end if
17:        end for
18:        n.queueUpstreamOut.put(t)
19:    else if m.type == ACK then
20:        message t
21:        invalid = 0                            ▷ Count all invalidated candidates
22:        for datapoint c in n.candidates do
23:            if c.beingSent then
24:                c.beingSent = false
25:                if !c.valid then
26:                    invalid = invalid + 1
27:                    t.put(c)
28:                    dropFromCandidates(c)
29:                end if
30:            end if
31:        end for
32:        if invalid > 0 then
33:            t.type = INVALID
34:        else
35:            t.type = VALID
36:        end if
37:        n.queueUpstreamOut.put(t)
38:    end if
```

Algorithm 9 Transfer processing, Part 2

```
39:     m = n.queueDownstreamIn.get()
40:     if m.type == TRANSFER then
41:         for datapoint p in m do
42:             pos = couldInsertIntoCandidates(p, n)
43:             n.candidates.get(pos).beingReceived = true
44:         end for
45:         message t
46:         t.type = ACK;
47:         n.queueDownStreamOut.put(t)
48:     else if m.type == VALID then
49:         n.sentHasSpace = false                    ▷ We can request the next transfer
50:         for datapoint c in n.candidates do
51:             if c.beingReceived then
52:                 c.beingReceived = false
53:             end if
54:         end for
55:     else if m.type == INVALID then
56:         n.sentHasSpace = false
57:         for datapoint c in n.candidates do
58:             if c.beingReceived then
59:                 c.beingReceived = false
60:             end if
61:             dropFromCandidates(c)
62:         end for
63:     end if
64:     if !n.sentHasSpace and n.candidates.freeCount > transferlimit then
65:         message m
66:         m.type = HAS_SPACE
67:         m.put(n.candidates.freeCount)             ▷ How many we can take at least
68:         n.queueDownstreamOut.put(m)
69:         n.sentHasSpace = true
70:     end if
71: end procedure
```

Case c) *No point in transfer.* The two points will be compared eventually when one of them is accepted into a working set with timestamp t_i. After their comparison, the other point is either also accepted into the working set (in which case they are not compared any more) or is put into the overflow with a timestamp t_j. Because of the strictly increasing nature of the timestamps, $t_j > t_i$. The point in the working set will be removed eventually, but at the latest when the two points meet again: Then the output condition is satisfied and the point in the working set is output without evaluation of their dominance.



(a) A successful transfer

(b) An transfer with invalidation

Figure 4: Occurrence of VALID and INVALID in the transfer protocol

5.2 Optimizations

This section details optimizations which have been implemented in the Skyline Shifter List algorithm. The impact of some of these implementations is measured in section 6.

5.2.1 Linked list working set

The working set was initially implemented as an array of fixed size, with a flag indicating if this array element was occupied or not. The size was a compile time constant, which did not allow much flexibility to configure the windows. Dynamic allocation with a configuration value eased this problem.

Because finding the next 'free' element in an array is an operation with linear complexity (and this operation is needed when inserting new data points into the data set), we switched to a linked list implementation with doubly linked elements and a pointer to the last element. The list elements are allocated and deallocated on demand.

The linked list allows insertion to be performed in constant complexity, however we lose the advantage of spatial locality as well as possible compiler optimizations related to arrays (especially, if the window size is a compile time constant).

5.2.2 Load balancing; Working set scaling

The concept of Shifter List relies on the parallel processing of many elements, using pipeline parallelism. This pattern exhibits the best performance if all units of the pipeline are busy and operating at the same speed (no bottlenecks).
Because in our Shifter List implementation every node eliminates points, the further down the pipeline, the less input is received by a node. We have noticed that this imbalance leads to significant performance hits.
To ensure better load balancing, we have introduced working set scaling: The initial node has the smallest working set, with subsequent nodes getting increasingly larger windows. This approach is based on the assumption that smaller working sets can contain less points which in turn leads to less points which are dominated by this window.

We expect that this optimization will have the highest effect on data sets with a medium to large percentage of skyline points with respect to the input size. It will have no effect on very small skylines that fit into the first working sets: These will eliminate most points and leave insufficient load for the subsequent nodes.

5.2.3 NUMA aware scheduling and data placement

Our implementation pins threads to specific cores using the *pthread* library call `pthread_attr_setaffinity_np` to avoid thread migration and use the fact that we know the special communication and data sharing pattern of Shifter List in advance. We pin one thread to each physical processor core before starting to assign threads to an already used core and forcing the operating system to schedule them.

Because communication channels between nodes are shared and accessed frequently, we schedule threads in core-major order (as recommended in [18], chapter 3.1.2). This assignment of threads implies that we first 'fill up' a NUMA region with threads, before proceeding to the next

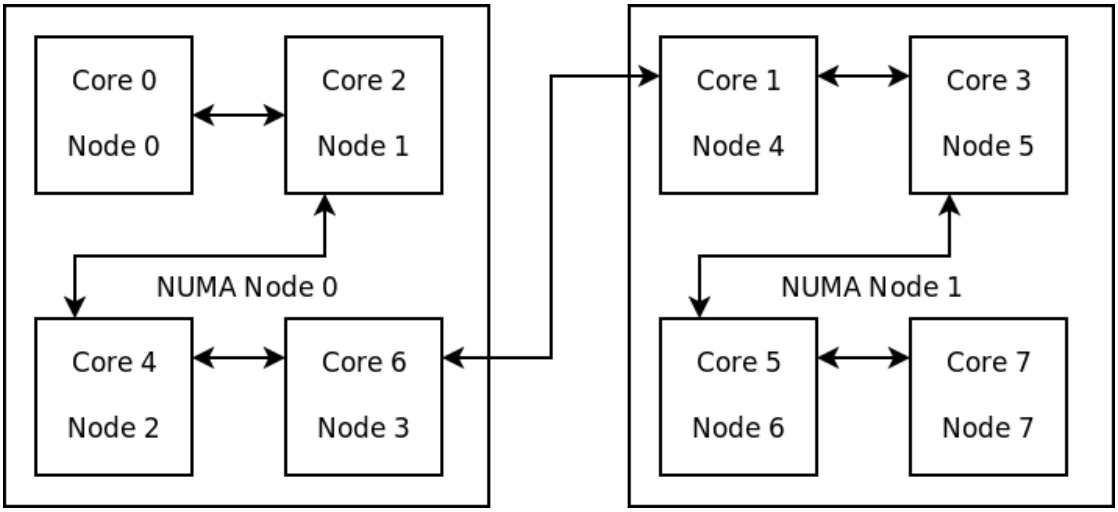


Figure 5: Example of optimized scheduling on a system with 2 NUMA regions and 8 logical processors. Communication channels have been collapsed into bidirectional channels for simplicity.

one. This assignment also guarantees the lowest number of communication channels crossing the NUMA node boundaries, while still utilizing all cores (see Figure 5).

In the optimal case, most nodes actually are located on one physical processor socket as their neighbouring node(s).
The default policy of Linux of ordering its logical processors, as we observed it, seems to follow a core-major ordering. Following this order (pinning threads to sequentially increasing logical processor ids) the threads would be allocated in a round-robin fashion on the nodes, leading to each communication channel crossing a NUMA boundary (for $N > 1$ NUMA regions).

The data placement is unchanged from the default allocation policy of Linux. However, we take care to allocate the communication channels and working sets on the thread which is actually going to use them, taking advantage of the 'first-touch' allocation policy of Linux[4] [5]. This policy will only map the virtual memory to physical memory once it is actually used (written/read). The physical memory will belong to the node that touches it first. Thus it is important to allocate the memory on the right node.

To discover the NUMA topology we use the *numa* library[6] and then pin the threads to the appropriate logical processors.

When some processors offer Simultaneous Multi-Threading (SMT, e.g. Hyper-Threading), the operating systems sees more logical processors than physical processors exist. To avoid resource contention, it may be desirable to avoid these virtual logical processors. The implementation accepts a bitmask to forbid certain logical processors. Otherwise our thread pinning implementation would inadvertently pin threads to physical and virtual processors that map to the same physical core.

[4] `http://linux.die.net/man/2/set_mempolicy`, retrieved on 2013-06-06
[5] `https://www.kernel.org/doc/Documentation/vm/numa_memory_policy.txt`, retrieved on 2013-06-06
[6] `http://oss.sgi.com/projects/libnuma/`, retrieved on 2013-06-07

5.2.4 Start phase: Eager accept

In the initial algorithm implementation only the last node accepted data points into its candidate set. This however led to a high number of transfers and the initial nodes basically only forward the data. Further, this implementation showed a high number of elements that needed to be passed on because of a full working set at the last element: Remember that the transfer protocol needs every transfer to be acknowledged before the sender can free the space in the local working set again. Thus we adjusted the algorithm slightly, this was inspired by the FPGA implementation (state automaton in figure 10, [14]).

A node can accept a data point into its candidate set if and only if:

- They are not dominated by any data point in the local candidate set, and
- The node's candidate set is not full, and
- The node is
  a) The last node, or
  b) Has never had a full local candidate set

Algorithm 10 Accepting elements into the candidate set (Optimized)

```
1:  procedure couldInsertIntoCandidates(datapoint pt, node n)
2:      if n.candidates.full() then                    ▷ Cannot insert in any case
3:          return -1
4:      else if n.isLast() or n.state == NEVER_FULL then
5:          pos = n.candidates.put(pt)
6:          if n.candidates.full() then
7:              n.state = FULL
8:          end if
9:          return pos
10:     else
11:         return -1
12:     end if
13: end procedure
```

The only change is that nodes which have never had a full candidate set may also accept data points into the candidate set. It is trivial to see that this does not violate correctness: If a node n_i has never had a full candidate set until a time t, it will not have forwarded any data points before t. Thus there are no data points at nodes further downstream: The data point has been compared against all input so far and can be accepted into the candidate set.

5.2.5 Challenges

After initial measurements we discovered that a vital point in the performance of the Shifter List implementation is the time of the initiation of the transfer protocol. Our initial implementation only transferred points to the upstream node when the candidate set of the sending node was full - even if a part of the candidate set could already have been transferred. This delayed the transfers unnecessarily and led to a significant performance penalty.

5.3 Limitations

Our implementation currently uses three different implementations of the run loop to distinguish the initial, middle and last nodes respectively. Due to this implementation choice, we require at least three threads to run the algorithm.
This is an acceptable limitation as the current mid-range processors offers already 4 physical cores. It would be possible to remove this limitation, however at a significant increase in implementation complexity.

Further, our implementation for the input and overflow currently assumes that all input can fit into main memory. It could be adapted with a small effort to support other ways to store the overflow or stream the input.

Our implementation eagerly allocates respectively frees memory when data points are inserted or deleted. A custom memory management which caches these elements for later reuse may benefit spatial locality by allocating a contiguous chunk of memory.

6 Experimental data

In this section, we compare the performance of our skyline implementation discussed in section 5 against a state-of-the-art competitor: *pskyline*, developed by Park et al. [12]. Further we would like to investigate the behaviour of our Shifter List algorithm and determine the impact of our optimizations.
Unless noted otherwise, the algorithms allocated 32 threads, which is the maximum number of physical threads that our hardware supports without sharing computation resources[7], excluding shared caches.

Most experiments use one of the following three types of datasets. These distributions are typically used to evaluate skyline computation.

Correlated In a set of correlated data points, a data point which is 'good' in one dimension is also likely to be good in the other dimensions. In this dataset, the skyline size tends to be small.

Anti-correlated In a set of correlated data points, a data point which is 'good' in one dimension is likely to be 'bad' in the other dimensions. This will result in a larger number of incomparable data points and thus in a larger skyline. In this dataset, the skyline tends to be of large size.

Uniform In a uniform data set, data points are chosen from a uniform distribution. For such a distribution, every possible combination has the same likelihood to be observed. Unlike in the former two data sets, this means that different dimensions of a data point are independently chosen at random.

[7] As in Hyperthreading by Intel processors, or sharing Floating Point Units by AMD Opteron processors.

6.1 Optimization impact

6.1.1 Array vs. linked list with scaling

We evaluated the implementation of the working set as an array and compared it to the implementation using a linked list.

The array implementation used 3 different sizes to allow for some working set scaling: One size for the initial node, one for the last node and one for the remaining nodes in between. These sizes were configured as (16, 128, 1024) and (512, 2048, 4096), respectively.
The linked list implementation used a working set scaling factor of 1.5, an initial working set size of 2 and a maximum working set size of 2^{20}. The data set contained 102400 points and were distributed according to an anti-correlated distribution.

In Figure 6a, we plotted the runtime of the skyline algorithm for both implementations. We can observe that the best configuration of the array is comparable - and for the dimensions 12 and 14 even better - than the linked list implementation. The array configuration with a working set size of 16 for the initial node did show a very bad performance. This is likely due to the too small overall working set size which leads to many iterations in our algorithm in which points have to be written and re-read from the overflow.

To see if this performance still holds for larger data sets and skyline sizes, we ran the experiment with an input size of one million points and an anti-correlated distribution. In Figure 7a and Figure 7b, we can see that the linked list now clearly outperforms the array implementation.
While the mediocre performance of the linked list on the smaller data set was unexpected, there are four main reasons why this is the case:

1. The array version exhibits very good spatial locality, being highly efficient for the processor to iterate over the working set when implemented as an array.

2. The linked list exhibits very poor spatial locality ("pointer chasing") which could be worsened by the eager allocation mentioned above.

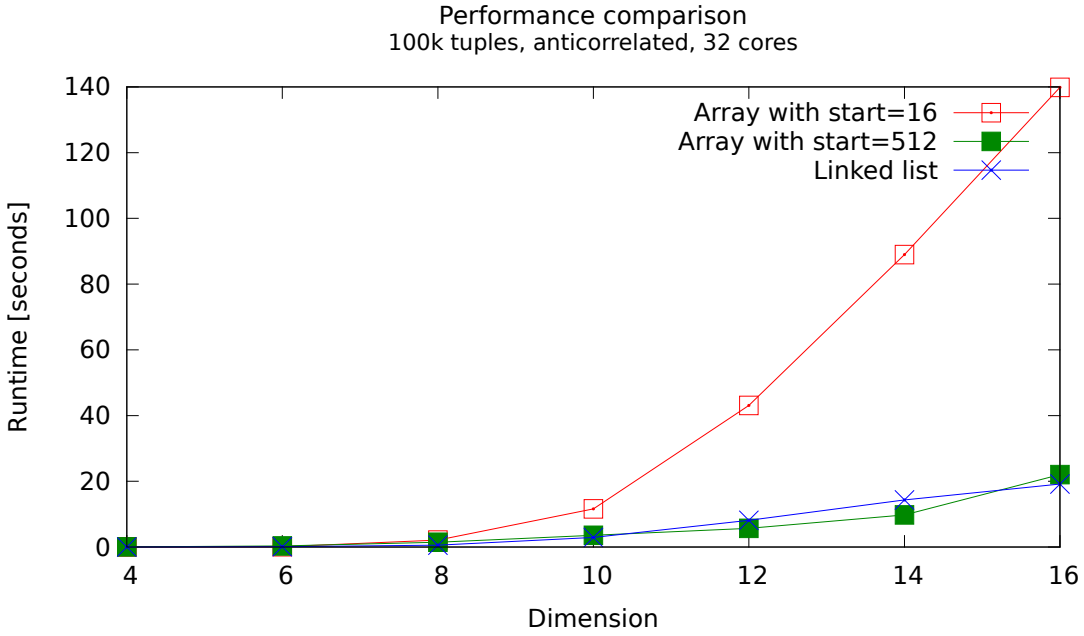
3. The working set size was a compile time constant - allowing an abundance of optimizations by the compiler.

4. The linked list eagerly used *malloc* and *free* when dropping or adding an element from the list.

With the higher performance on the larger data sets and the increased flexibility of the linked list we will continue to use this version. Possible optimizations that manually manage the memory allocation of the linked list may prove beneficial for performance.

6.1.2 NUMA scheduling

This experiment examines how the impact of the custom scheduling and thread pinning is, opposed to specifying nothing and letting the Linux scheduler take care of scheduling.
To achieve this, we call `pthread_create` without setting any processor affinity before or after the threads start. The experiment was executed on a 4 socket machine equipped with four AMD Opteron 6276 processors. Such a processor actually consists of two combined full processors ('modules') connected via a HyperTransport link. Each module has 6MB of L3 cache. We made



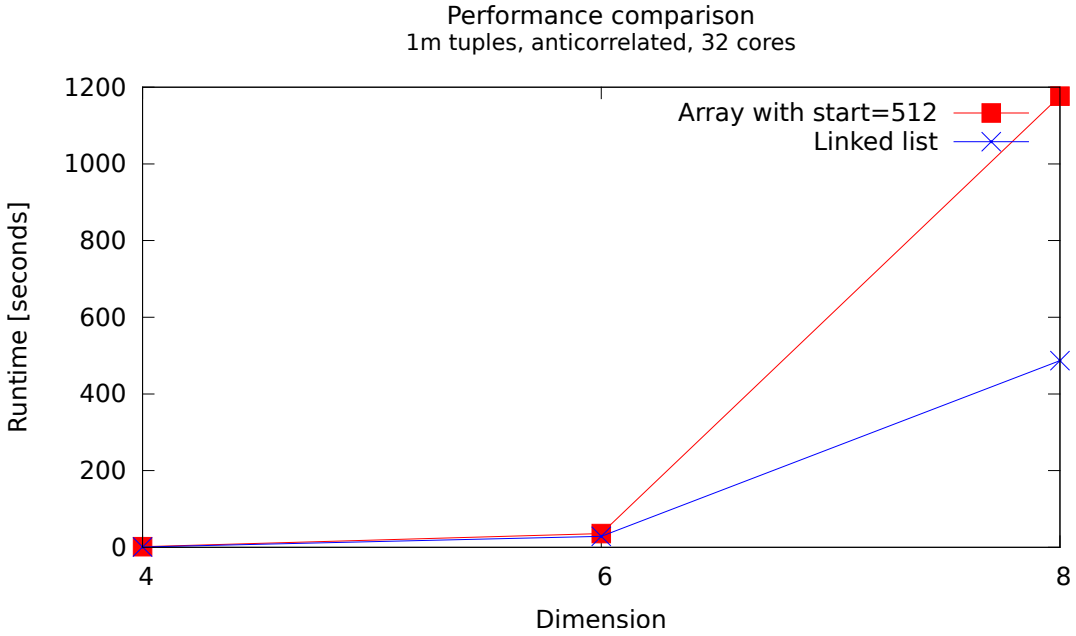
(a) Array vs. linked list performance

	4	**6**	**8**	**10**
Linked list	0.077	2.312	9.001	14.958
Array (start=16)	0.131	9.391	55.567	102.554
Array (start=512)	0.408	3.338	5.792	10.757

	12	**14**	**16**	
Linked list	15.470	16.106	15.322	
Array (start=16)	127.240	129.478	130.285	
Array (start=512)	14.467	17.410	16.959	

(b) Array vs. linked list runtimes in seconds

Figure 6: Performance of Array and Linked List implementation



(a) Array vs. linked list performance with larger data set

	4	**6**	**8**
Linked list	0.642	28.627	487.033
Array (start=512)	1.659	35.642	1176.618

(b) Array vs. linked list runtimes in seconds, with larger data set

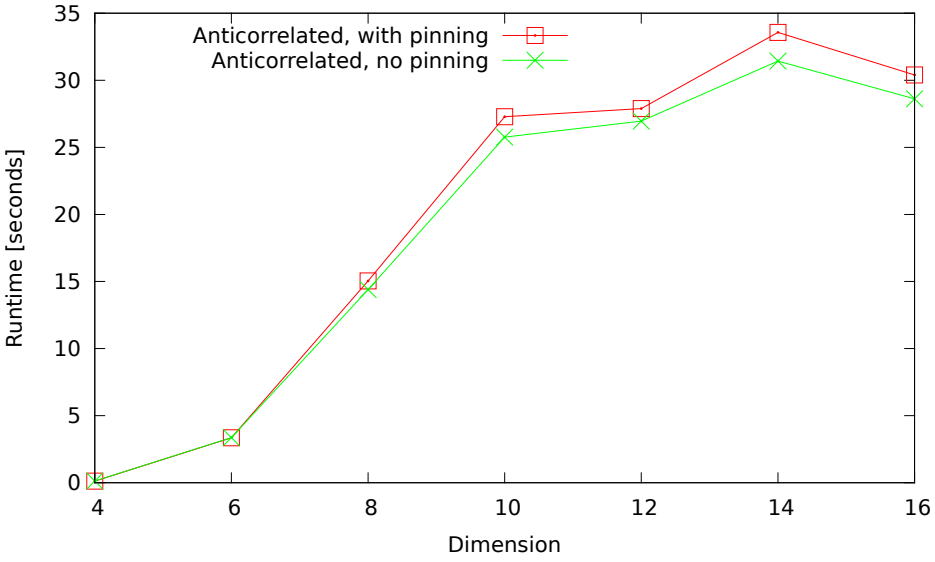


Figure 7: Impact of NUMA aware thread pinning

sure that logical CPUs which share certain resources[8] are not both used.

We hypothesize that our variant which pins the threads to the cores as we specified in subsubsection 5.2.3 performs better or at least as good as the variant which leaves the thread placement up to the operating system, since we know the communication pattern of our threads in advance.

However, the data we obtained does only partly match our hypothesis: For the uniform point distribution, our performance when using pinning is almost identical to the one without pinning. In the anti-correlated case plotted in Figure 7, we are even slightly slower when using the pinned version.

In our experiments regarding NUMA-awareness we have used both the knowledge of the NUMA configuration and the communication pattern of our algorithm. However, we did not achieve performance benefits with this knowledge. We assume that other effects outweigh and obscure the effects of NUMA.

6.1.3 Working set scaling

As mentioned in subsubsection 5.2.2, we use scaling of the working sets to ensure sufficient load at nodes further downstream. In this experiment, we evaluated the effectiveness of this optimization.

To evaluate this, we ran a query and evaluated the number of points that were not forwarded by a node. These points were either dominated, added to the working set, or output. This measurement does allow us to detect nodes that have no work or only a low load, however does not give us a utilization value. To ensure proper results, the skyline must be uniformly distributed among the input points.

[8] Instruction cache, instruction decode logic, floating point unit. See [19], p.5

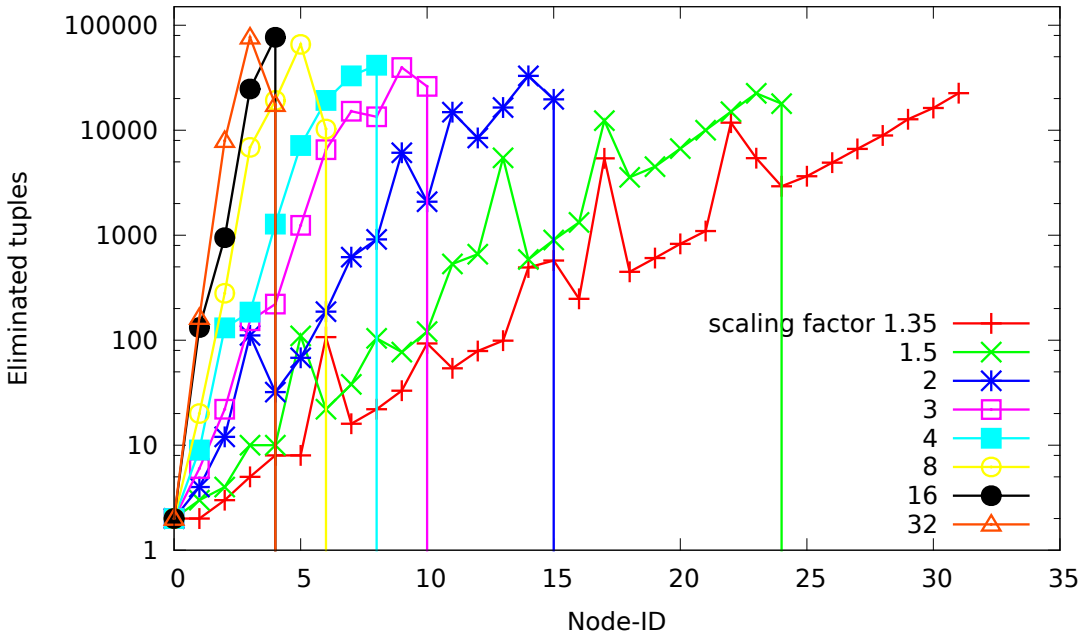


Figure 8: Load balancing with different working set scaling factors

Setup We use an initial working set size of 2 for the first node and then multiplicatively scale this by a constant factor up to a maximum working set size of 2^{20} elements per node. In this experiment, we vary the working set scaling factor. We used an anti-correlated distribution to generate the dataset of 102400 tuples and 16 dimensions per tuple.

Data In Figure 8 we plotted for each working set scaling factor the number of eliminated tuples per nodes. The x-axis reflects the sequence of the nodes in the Shifter List, the initial node being at x=0. We can see that for all scaling factors except 1.35, only some nodes perform any action while the rest are not used (easily recognizable by the drop of the line to zero).

In Figure 9 we plotted the runtime needed for each working set scaling factor.

Analysis We observe that all curves have a rising characteristic, which is expected given that we increase the working set for each node.

We can clearly see by the steep curves in Figure 8 that the higher the scaling factor is, the more load is placed on the first nodes. After a few nodes, there is no more load left for the remaining nodes. The extreme case is the factor 32: Only four nodes do actual work.
This is contrasted by the factor 1.35: All nodes have work to perform and the curve almost approximates a linear curve in the log-plot (which matches the characteristic of our working set scaling).

In the performance plot in Figure 9, we observe that our working set size scaling has a significant impact on the runtime of our algorithm. The scaling factor of 1.35 shows the best performance which confirms our hypothesis that we need to use the potential of all nodes to achieve best performance.

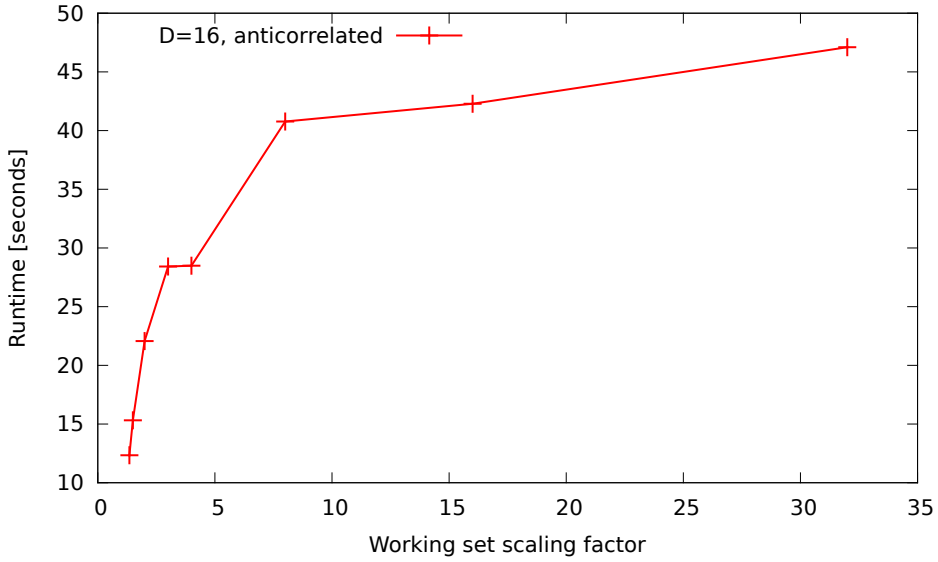


Figure 9: Load balancing effect on performance

6.2 Skyline size

In this section we analyse the performance on three different workloads: Anti-correlated, correlated and uniform datasets.

6.2.1 Setup

We used the data generators of *pskyline*[12] to generate these data sets. All values in this experiment are, unless mentioned otherwise, medians of 5 repetitions.

In all the datasets we evaluate different selections of the windows scaling factor and compare it against *pskyline*. All implementations had 32 threads assigned. The measurements were performed on a 4-socket Intel Xeon E-5 (E5-4640) Linux machine with 20MB of L3 cache per Socket. Hyperthreading was enabled and the system provided 32 physical and 64 logical processors respectively.

The measured runtimes for *pskyline* were taken from the reported time in the output, which excludes the time to load the data file from disk. For the Shifter List implementation, the measuring was performed analogously.

6.2.2 Anti-correlated

In Figure 10, we can see significantly better performance of the Shifter list implementation compared to *pskyline* in both higher and lower dimensions. The performance is better by a factor of 8 to 15 for this dataset.
At $D > 12$, we observe a stabilization of the performance for both implementations, as the size of the skyline does not grow any more and actually decreased slightly in the generated dataset.

	4	6	8	10
Pskyline	1.1966	23.3411	81.8337	128.117
Shifter List(Best)	0.0773	2.3124	9.0014	14.9583
	12	**14**	**16**	
Pskyline	144.4039	140.978	134.3983	
Shifter List(Best)	15.4703	16.1058	15.3217	

Table 1: Runtime of *pskyline* and best Shifter List on anti-correlated distribution

Further we observe that the smallest working set scaling factor (1.5) exhibits the best performance among the Shifter List implementations.

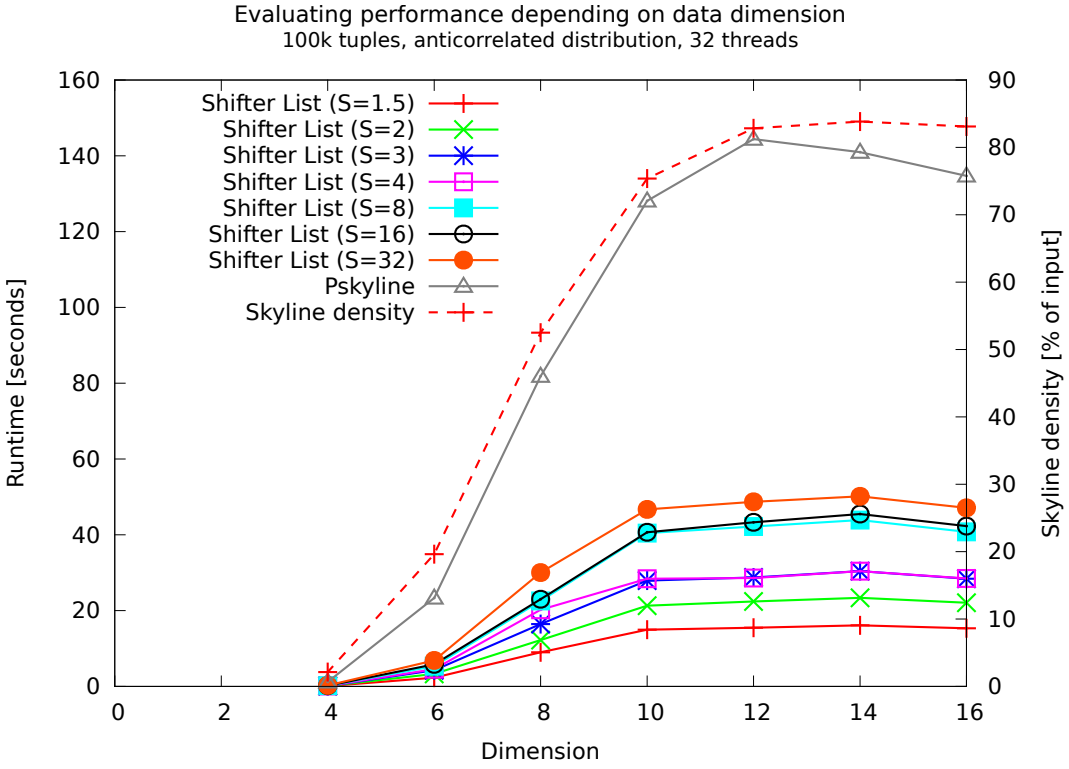


Figure 10: Performance evaluation of a anti-correlated distribution

6.2.3 Uniform

As we can see in Figure 11, our implementation significantly outperforms *pskyline* in the higher dimensions which deteriorates quickly for D>8. However, we also outperform *pskyline* in smaller dimensions, as can be seen in Table 2.
Overall we can see a performance increase of a factor of 3-10.

	4	6	8	10
Pskyline	0.0444	0.8631	6.901	27.2466
Shifter List(Best)	0.0144	0.0797	0.5737	2.9207
	12	**14**	**16**	
Pskyline	69.8414	123.0952	163.79491	
Shifter List(Best)	8.1504	14.3108	19.1508	

Table 2: Runtime (in seconds) of *pskyline* and the best Shifter List configuration on the uniform data set

As before, the configuration with a working set scaling factor of 1.5 was the best performing implementation among the Shifter List measurements.

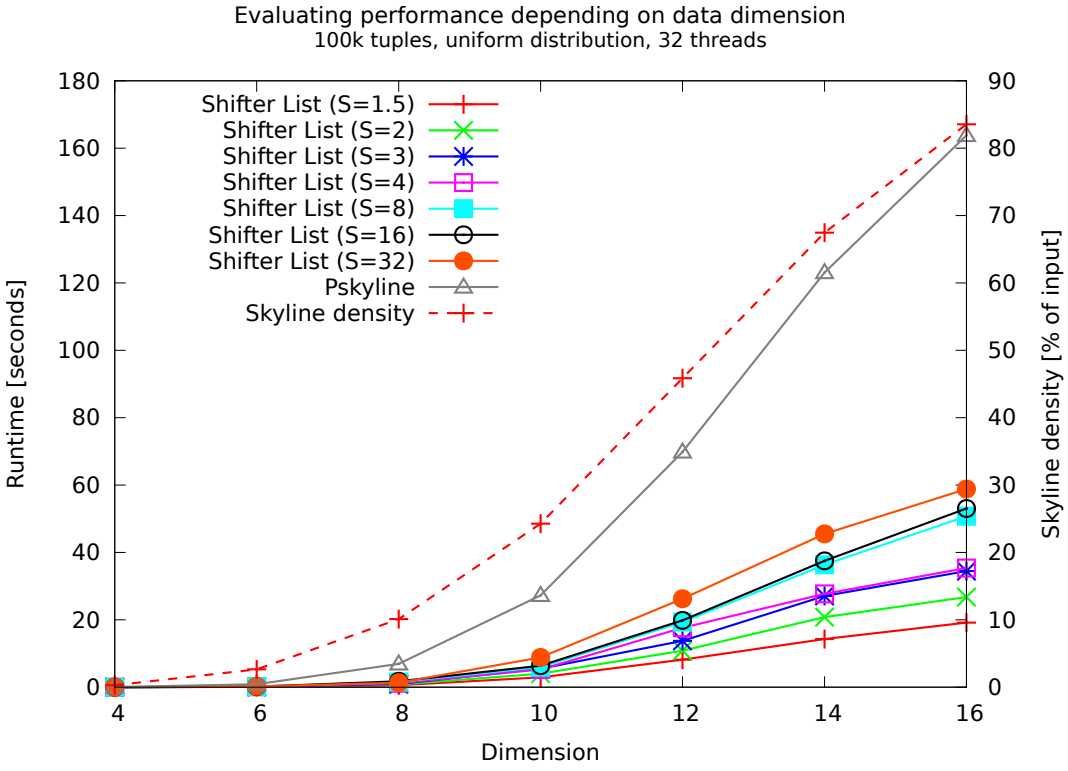


Figure 11: Performance evaluation of a uniform distribution

6.2.4 Correlated

For the correlated dataset, where the skyline is very small (less than 0.01% of the input set), we can see in Figure 12 that the performance of *pskyline* is better by about a factor of 1.3 to 2

compared to the best Shifter List configuration.

Moreover the best Shifter List implementation is the one with the largest working set scaling factor, which indicates configurations in which the first nodes process all input are preferred[9]. This could indicate that multi-core approaches in general are not a good fit for this kind of dataset where the skyline is small with respect to the input set. The same conclusion was drawn by Köhler et al.[10] To corroborate this theory, we have added a single-threaded naive implementation for comparison to Figure 12: We can immediately see that the naive implementation far outperforms both implementations. On datasets with very small skylines, other approaches would thus be more appropriate than these multi-threaded approaches.

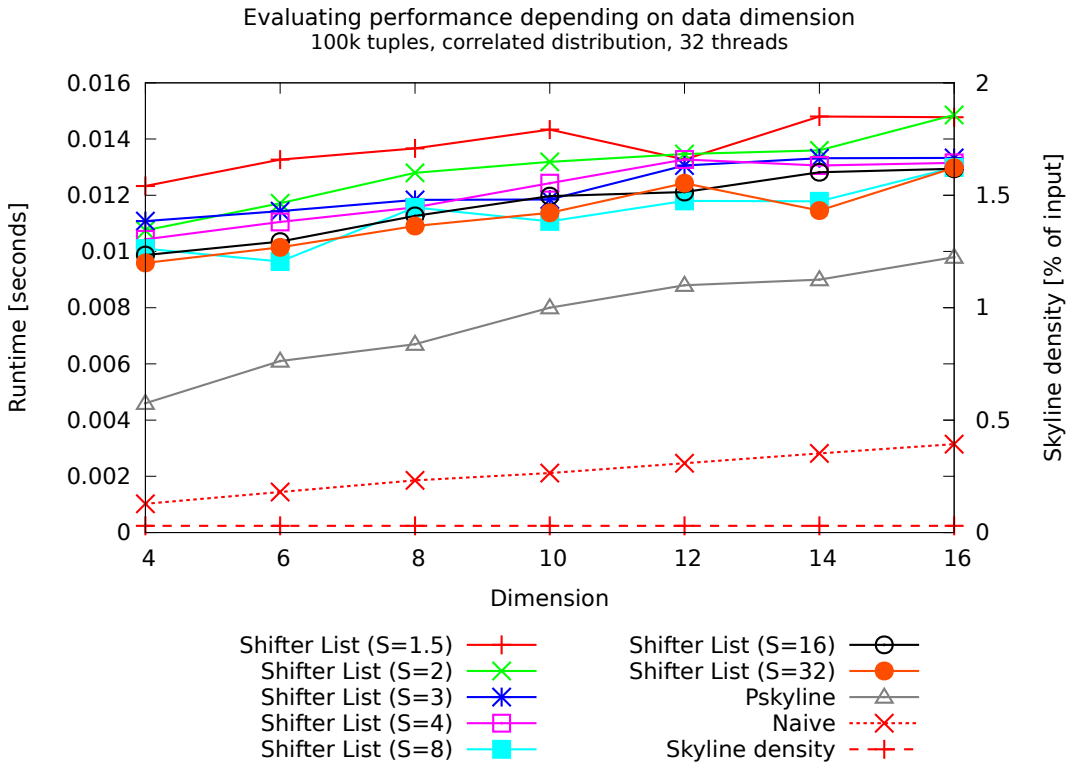


Figure 12: Performance evaluation of a correlated distribution

6.3 Data skew

We evaluated how our algorithm performs if the position of the skyline elements does not follow a uniform distribution within the dataset. To do so, we evaluated datasets where the skyline elements are clustered with varying density at the beginning of the input and at the end of the input.

[9]The larger the window, the more data points are accepted into the candidate set in the eager start phase.
[10]See [11], page 91

	4	6	8	10
Pskyline	0.0046	0.0061	0.0067	0.008
Shifter List(Best)	0.0096	0.0101	0.0109	0.0114
	12	**14**	**16**	
Pskyline	0.0088	0.009	0.0098	
Shifter List(Best)	0.0124	0.0115	0.0130	

Table 3: Runtime (in seconds) of *pskyline* and the best Shifter List configuration on the correlated data set

6.3.1 Setup

The dataset was generated by the *pskyline* generator. After completion, we ran our implementation to determine which points belong to the skyline and which do not and split the data into two files. A small script then merges these two types of points into a single data set with the requested stride size between the skyline elements: A distance of zero means that for a skyline size of N, the first N elements of the dataset belong to the skyline, a value of one that every second element belongs to the skyline. We steadily increase the distances as to evaluate how resistant the algorithms are to skew. Further we repeat the same experiment with the whole dataset reversed, i.e. the skyline spaced from the end of the dataset.

The experiment was run on the same 4-socket Intel Xeon machine as the previous experiment. We used a target skyline size of slightly more than 12'000 elements from an input data set of approximately 10 million elements, which is a skyline density of slightly less than a thousandth of the input size. Both implementations had 32 cores available. The Shifter List implementation was configured with a working set scaling factor of 1.35.

The stride size was varied from zero to 834. With the latter stride size, the elements are spaced such that they span the entire input. Increasing the stride size any further is not possible without reducing the number of skyline elements.

6.3.2 Hypothesis

Because *pskyline* operates according to an input-partitioning scheme, we expected a stronger skew would lead to a worse performance. This due to the assumption that when the skyline elements were close together, only few input partitions would contain skyline elements and the remaining partitions would contain many data points that did not belong to the final result. We assumed that this would lead to unnecessarily large intermediate results and require more work in the reduce step.
As Shifter List greedily accepts elements into the input set at the beginning, we expected that a skew to the beginning would lead to the best performance, while a uniform distribution or a skew toward the end of the dataset would lead to more data points being put into the overflow - and thus a worse performance.

6.3.3 Results and analysis

The performance results can be seen in Figure 13a and Figure 13b, the reverse data set results in Figure 14a and Figure 14b. We observed that both implementations are influenced by skew

in the dataset.
Pskyline behaved as hypothesized: If the dataset exhibited heavy skew (whether toward the beginning or end of the dataset) the performance is worse than when the skyline elements are uniformly positioned in the dataset (at stride size of 834). If the skew was toward the end of the dataset, then *pskyline* was even more affected: The performance decreased by a factor of 5 if the stride size was changed from a uniform spacing of the skyline elements (stride 834) to a dense clustering at the end of the data set (stride 0).

The Shifter List-based implementation behaved partly as expected in our hypothesis: For a heavy skew at the beginning, it outperformed *pskyline* significantly by a factor of 74. For the higher stride sizes, the performance then worsened. For the skew toward the end of the dataset, we did however observe a similar behaviour: The heavier the skew, the better the performance got. That our implementation performs better for a skew toward the beginning than the end can be explained by the eager accept optimization which clearly favours the skew toward the beginning. The performance improvement for the high skew can be explained by an inspection of the statistics: For the skew towards the end with a stride size of 0, only a tenth of the transfers (2489) was needed compared to the stride size of 834 (24083 transfers).

Though the performance of both implementations are influenced by skew, our Shifter List-based implementation outperforms *pskyline* in any skew configuration explored in this experiment. Furthermore the impact of the skew on the Shifter List-based implementation is about equal to the impact on *pskyline* if the skew is toward the beginning. However the impact on *pskyline* is significantly higher if the skew is toward the end of the data set.

6.4 Scalability

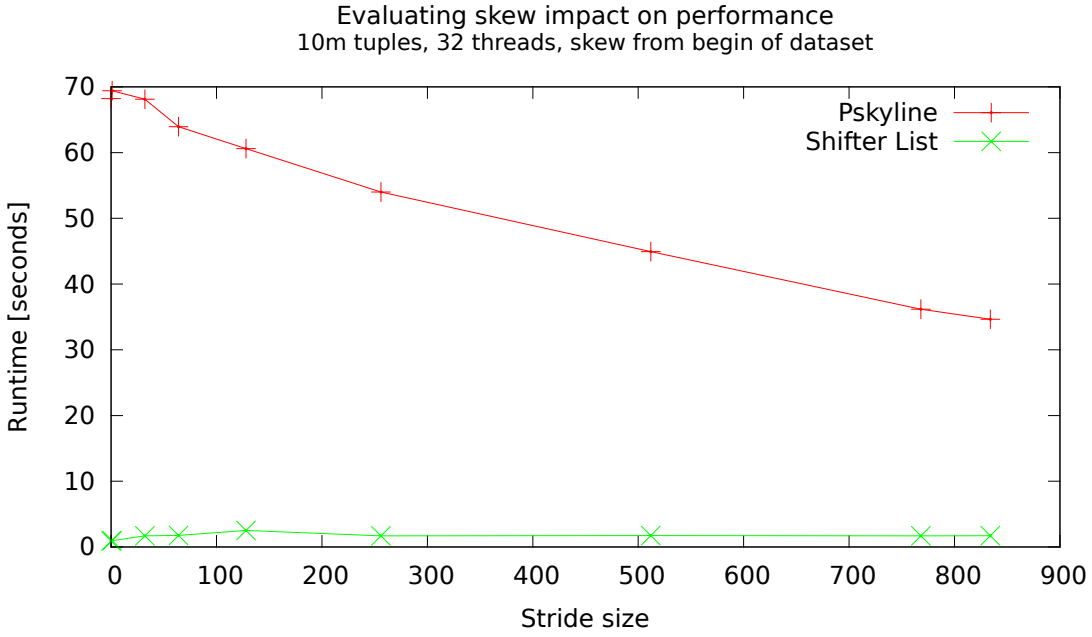
To evaluate the scalability of our algorithm, we vary the number of cores used for the algorithms and observe the impact on performance.

6.4.1 Setup

We used the linked list implementation with a working set scaling factor of 1.5. Because reducing the number of nodes would also reduce the working set size, we made sure that the overall size of the working set was always (approximately) equal by adjusting the working set size of the initial node (and adjusting the maximum working set size if necessary). This effectively increases the working set size per node but keeps the overall size constant.
Without such an adjustment, the different working set size would lead to significantly different results as the Shifter List would need different numbers of iterations to processes the data (as less points can be held in the working set per iteration).
This experiment was run on our Intel machine (the hardware details are mentioned in subsection 6.2).

6.4.2 Data and Analysis

We plotted the runtime of the algorithms vs. the number of allocated cores in Figure 15. We observe in the figure that both *pskyline* and our implementation improved their performance with additional cores. Both implementations profited most when adding up to 16 (Shifter List) resp. 24 cores (*pskyline*). After this point, the performance of *pskyline* even degraded, while the performance of our Shifter List-based implementation remained steady, and again improved

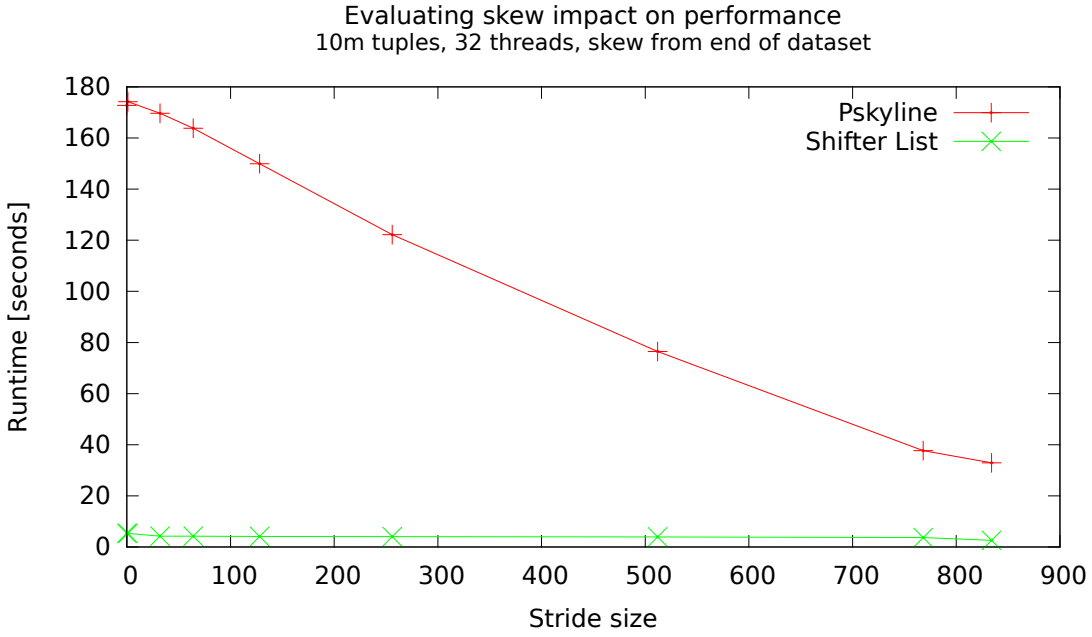


(a) Skew performance Shifter List vs. *pskyline*

	0	**1**	**32**	**64**	**128**
Pskyline	68.2252	69.4055	68.1215	63.9327	60.6038
Shifter List	0.9097	0.9621	1.6969	1.7729	2.5216
	256	**512**	**768**	**834**	
Pskyline	54.0099	44.9272	36.168	34.6405	
Shifter List	1.6867	1.7333	1.7019	1.7120	

(b) Skew performance (Runtime values in seconds)

Figure 13: Skew performance

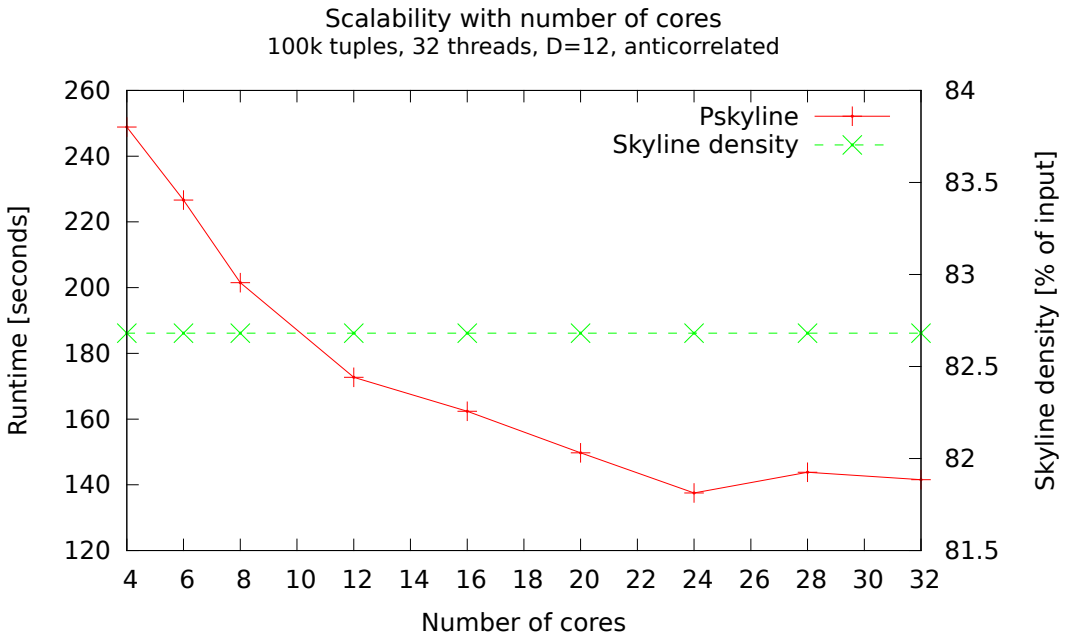


(a) Skew performance Shifter List vs. *pskyline* for reversed dataset

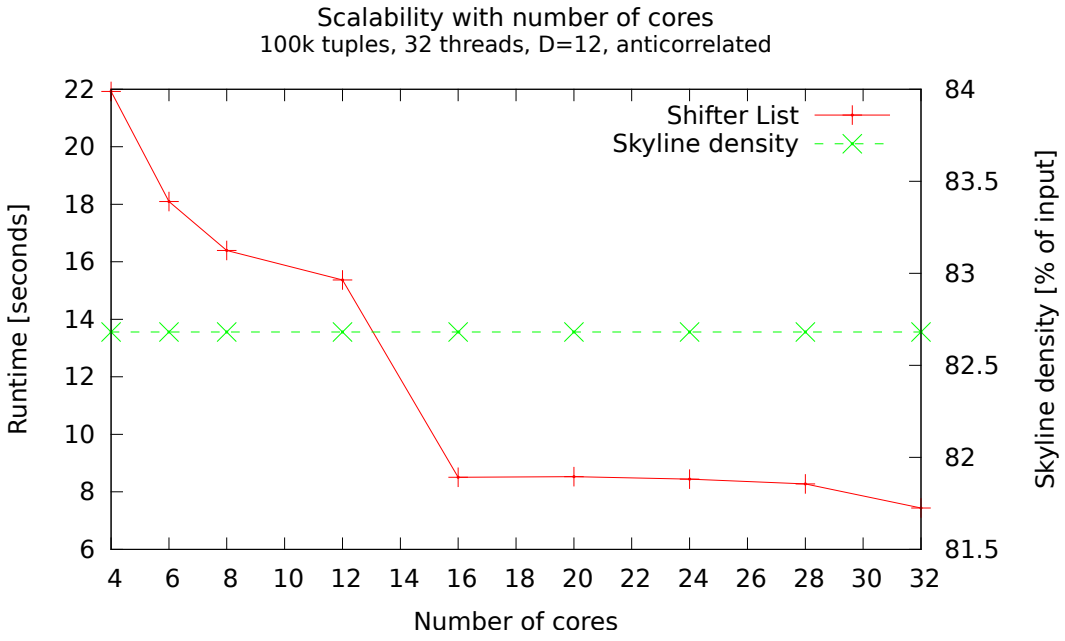
	0	**1**	**32**	**64**	**128**
Pskyline	172.7669	174.2273	169.6893	163.8039	149.906
Shifter List	5.3718	5.3036	4.2528	4.2033	4.0910
	256	**512**	**768**	**834**	
Pskyline	122.1815	76.4784	37.6953	32.9388	
Shifter List	4.0603	3.9069	3.6980	2.5730	

(b) Skew performance (Runtime values in seconds)

Figure 14: Skew performance for reversed dataset



(a) Scalability of *pskyline*



(b) Scalability of Shifter List implementation

Figure 15: Scalability of *pskyline* and Shifter List implementation

	4	6	8	12	16
Pskyline	248.88	226.63	201.53	172.73	162.36
Shifter List	21.92	18.09	16.37	15.37	8.51
	20	**24**	**28**	**32**	
Pskyline	149.74	137.51	143.80	141.5671	
Shifter List	8.53	8.44	8.28	7.44	

Table 4: Runtime values when adjusting number of nodes, in seconds. As plotted in Figure 15

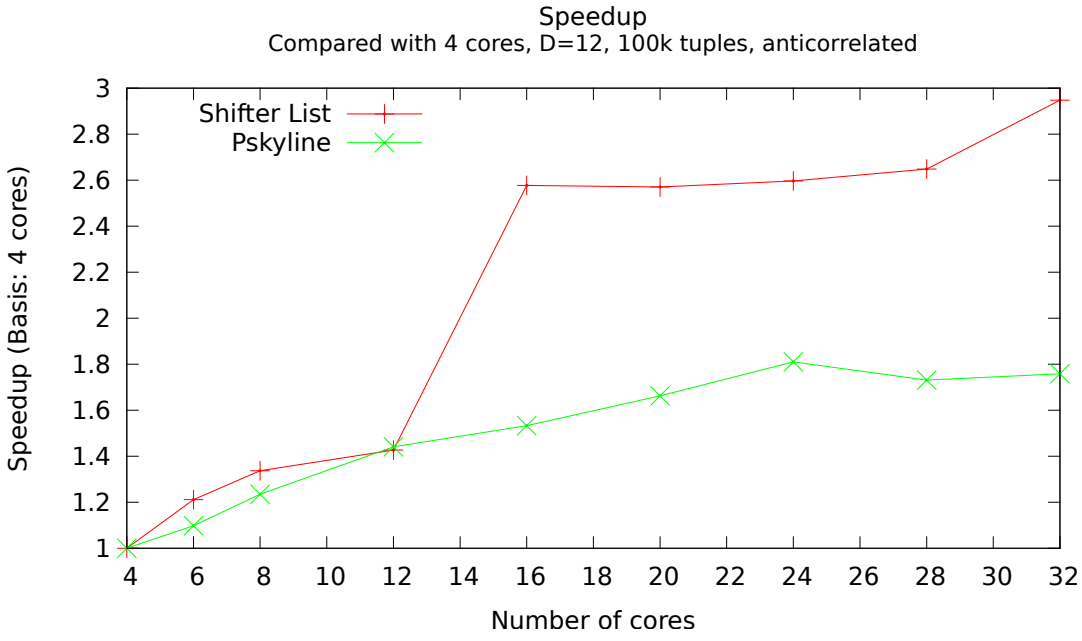


Figure 16: Speed-up of both implementations, compared with basis of 4 cores

again slightly when increasing the number of cores to 32.

We note that by going from 4 to 32 cores, the runtime of *pskyline* decreased by a factor of 1.75, while the runtime of Shifter List decreased by a factor of 2.95.

A significant jump in performance happens for our Shifter List implementation between 12 and 14 nodes: The performance almost doubles. To determine the cause, we re-ran the experiment but slightly enlarged the size of the working set, from approximately 85'000 to 100'000 elements. We plotted the comparison in Figure 17. We can observe that the second plot also has such a jump in performance at 24 to 28 nodes.

For the original setting with 12 nodes, the last three nodes had to handle half of the overall working set, which may have been too much. With 16 nodes, this load is spread out over 5 nodes. The same effect can be seen with the larger overall window: At 24 nodes, 4 nodes had to handle half of the working set compared to 5 nodes at 28 nodes.

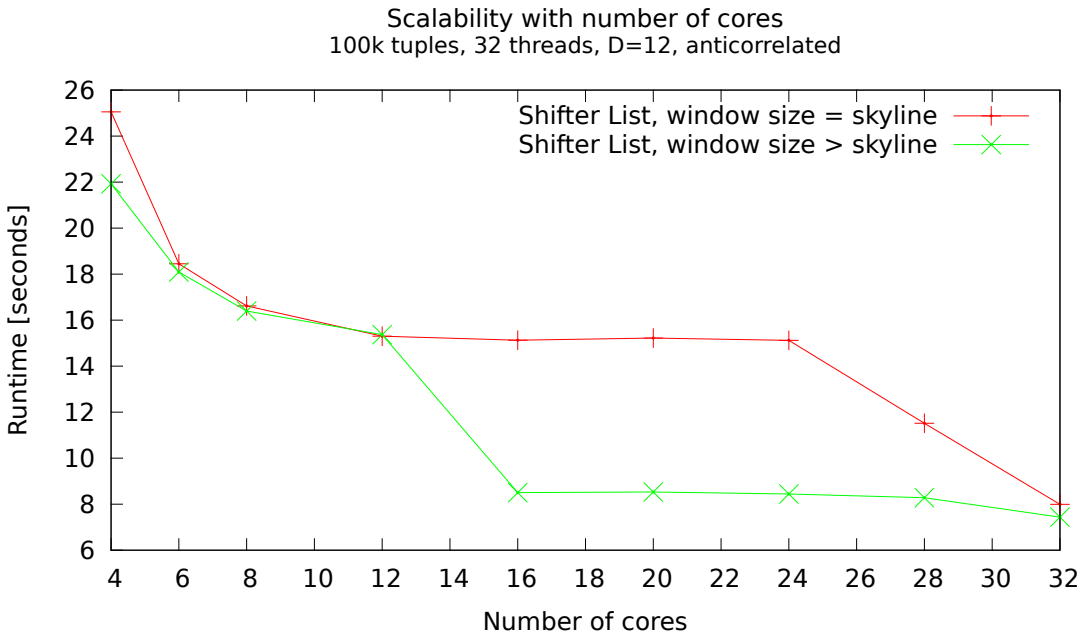


Figure 17: Comparison of Shifter List with two different working set configurations

We speculate that the load balancing should be configured to fit the number of available nodes. The constant scaling factor, which was originally determined for 32 nodes, does not necessarily guarantee a good load balance for less nodes. Either the factor needs to be tuned when changing the number of nodes or an automatic load balancing could be implemented.

6.5 Behaviour analysis

In this subsection, we will show some of the experiments that we have performed to evaluate and analyse the behaviour of our algorithm.

6.5.1 Runtime usage

In this experiment, we added more instrumentation code to our implementation to capture the execution time of different "tasks" that our implementation performs. These main tasks are: Processing points (testing for dominance), messaging (queue operations: send/receive), handling the transfer protocol and waiting for input. The remaining overhead (initialization, looping etc.) was ignored in this experiment. We used 102400 tuples with an anti-correlated distribution and executed it without any pinning or NUMA awareness, as this was shown to be ineffective by our earlier experiments. The experiment was executed on our Intel machine (see subsection 6.2 for hardware details).

In Figure 18 and Table 5 we plotted/listed the use of the algorithms runtime with regard to the identified tasks. Note that as some work is not captured by the measurements and the nodes do not start or end at the same time, the total runtime shown in the plot is not equal for each

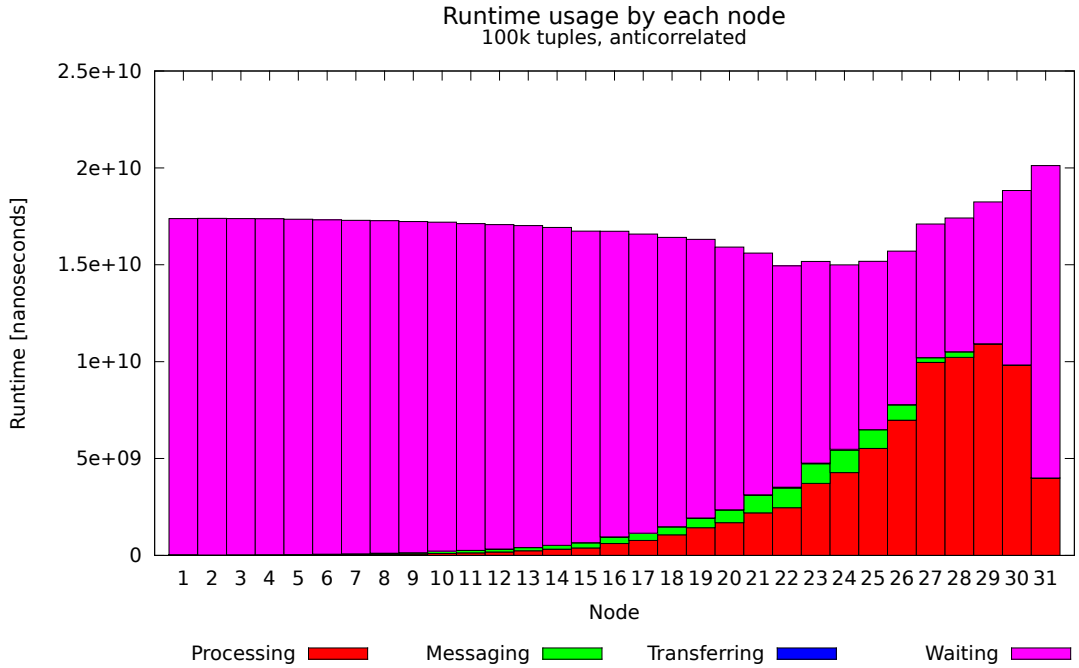


Figure 18: Runtime usage per core

Processing	**Messaging**	**Transferring**	**Waiting**
14.77%	1.91%	0.04%	83.29%

Table 5: Percentage used by tasks with respect to totalled runtime

node.

We can see in the plot that the messaging and transfer took only a very small part of the overall runtime of the algorithm. Further, we see that the processing took an increasing amount of the runtime the further downstream a node was located. This is caused by the increasing working set size: Every node has a working set which larger by a multiplicative factor than his predecessors (bounded by an upper limit). Due to this, the potential processing time needed per received data point increases which is reflected in the plot. We can observe that after Node 29 the processing part of the runtime dropped again, at which point so many data points were filtered out by the nodes upstream that the reduction in input data was stronger than the increase in size of the local working set.

Very prominent in the plot is the amount of "waiting" which accounted for the lion's share of the run time. It behaves inversely proportional to the processing time. This is also a consequence of the scaling of the working set: As initial nodes have a very small working set, they process their input elements received from the queues very quickly, thus incurring more wait time per received message (assuming a constant rate of messages) than their downstream peers. If a node drops or accepts all elements of a message into the working set, this leads to more wait time for subsequent nodes as this empty message will be discarded.

To break down the wait time further, we added additional instrumentation code. The first part of the waiting happens when a node's outward down queue is full: It will "stall" processing of messages containing data points. This is necessary as trying to put an element in a full queue will block until the operation succeeds (named "messaging wait"). If the node were to greedily process input messages and send the data elements downstream, this might result in a deadlock. The second part of the waiting time happens when the node does not have any input to process (no input from upstream nodes), we label this "waiting for input". We plotted the results in Figure 19. We can observe that the time for transfers and full queues ("messaging wait") are negligible. The messaging time is not negligible, but most interesting is the stalling: It accounts for a significant part of the runtime. This indicates that load balancing might be an issue which could improve performance further.

The high amount of stalling might indicate that the message queues between nodes are not large enough to compensate some fluctuations in processing speed of the different nodes. To confirm this influence, we increased the size of the queues and plotted both uniform and anti-correlated workloads in Figure 20 and Figure 21. We can see that stalling is no longer an issue.

6.5.2 Iteration/Working set size impact

In our skyline implementation we can output at most $M = \sum_0^{n-1} size(w_i)$ data points every cycle, where $size(w_i)$ is the size of the working set of the i-th node. This is due to our output condition that can only output data points from the working set. Should however M be smaller than the resulting skyline, some of the skyline points have to be put into the overflow file, requiring another iteration over the remaining input.
In this experiment, we examine how much influence the number of iterations has on the performance of our implementation. To do so, we vary the overall working set size while keeping our input set the same. We should note that cache effects could influence our results - the working set is a heavily accessed local data structure for each node.

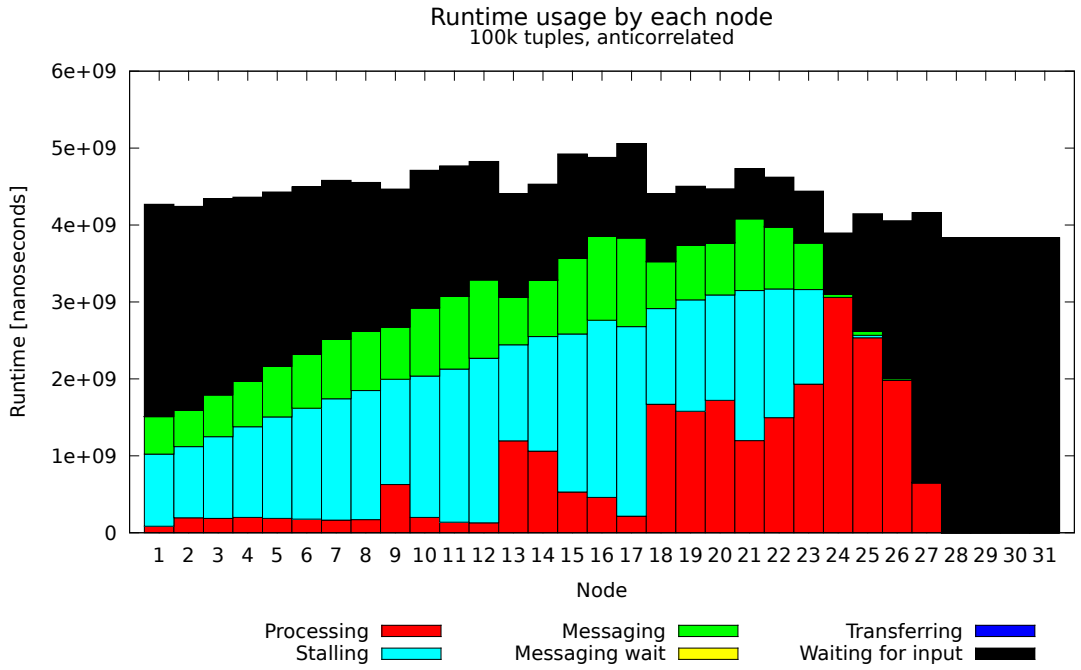


Figure 19: Detailed runtime usage per core

We used a data set of 102400 anti-correlated tuples with dimension 14 (the skyline size 85828 was known in advance). We used a working set scaling factor of 1.35 and used the maximum window size to control the total working set size. We varied the total size of the working set between 2% and 100% of the skyline size. We derived the lower bound for the number of iterations by dividing of the skyline size by the total working set size. The experiment was run on the Intel Xeon machine detailed in subsection 6.2.

In Figure 22 we plotted both the runtime and the number of iterations needed for a specific working set size. The number of iterations was plotted using a logarithmic scale. As expected, the number of iterations decreased rapidly as we increased the size of the working set.

The runtime decreased as expected starting at 20% of the skyline size. The data points below this size however, exhibited a lower runtime. This can be explained by cache effects: For the first two data points, the working set fit into the L1 cache of our machine, which greatly sped up the access and iteration over the set, offsetting some of the cost of additional iterations. The third data point thus lay higher again. The last anomaly is the fourth datapoint (20%) which lay unexpectedly higher, likely due to the same effect with the L2 cache (as with the L1 cache before).

Choosing the right size of the working set is thus very important for a good performance of our implementation. Choosing it too small will increase the number of iterations, choosing it too big will result in a bad load-balancing, leaving later nodes without work.

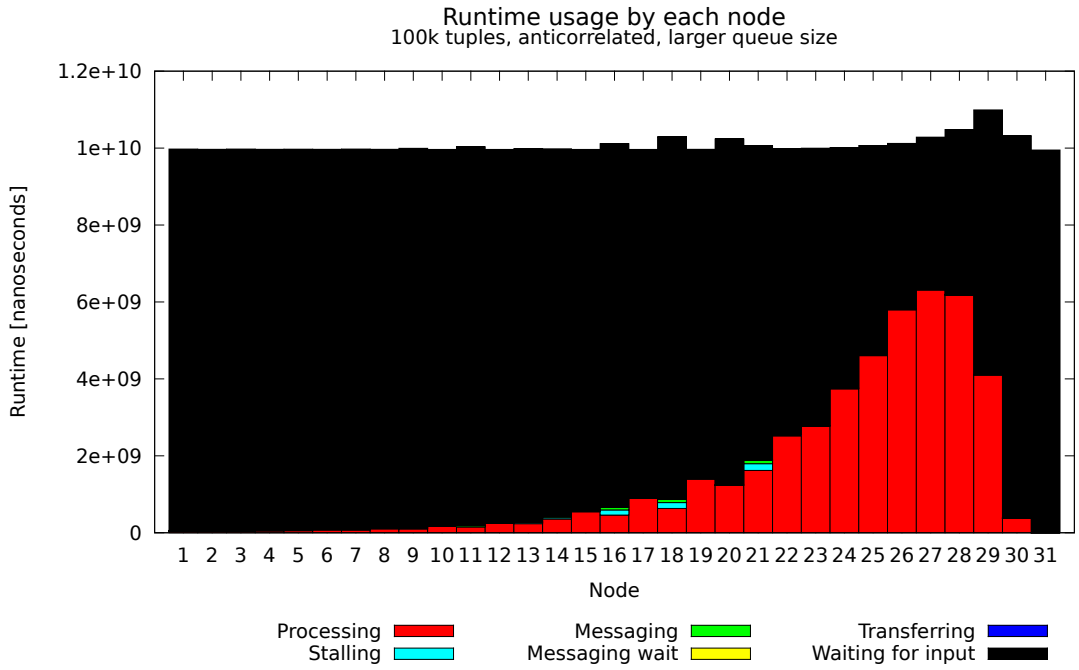


Figure 20: Detailed runtime usage with larger buffer queues (Anticorrelated)

6.6 Discussion

We have shown several experiments in this section. Our main findings include that our implementation outperforms *pskyline* by an order of magnitude on all but the correlated datasets (in which a naive, single-threaded performs best). Furthermore, we have shown that both our implementation and *pskyline* are affected by a skew of the skyline tuples in their distribution with the dataset (position), with our implementation being more resistant to skew toward the end of the dataset. Our implementation also showed a good speed-up when increasing the number of cores.
The other main finding is that the configuration of our implementation is vital to its performance.

6.6.1 Future work

Given the high influence of configuration parameters on the performance of our implementation, future work should develop a load balancing solution to fully ensure an even distribution of load across nodes and to minimize the amount of waiting or stalling at the nodes.
Such an approach could influence the size of the working sets at runtime. Increasing the working set size at upstream nodes if a node has too much load will result in a higher number of comparisons against working set elements for these nodes, eliminating more tuples from the data stream and resulting in less load for downstream nodes. Decreasing the working set on the other hand can be used to push more load toward downstream nodes.
The load balancing will have to provide a heuristic to detect the conditions for initiating the re-balancing and ensure that enough time is provided for transfers to take place and shift the working set items accordingly. The scheme has to avoid situations where the load balancing is

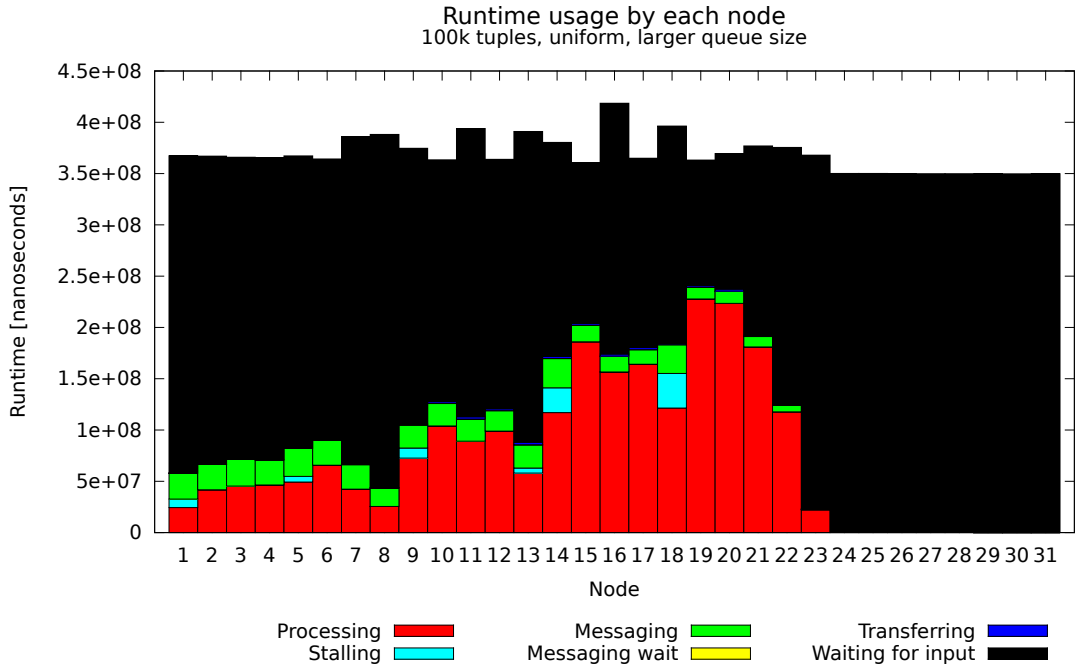


Figure 21: Detailed runtime usage with larger buffer queues (Uniform)

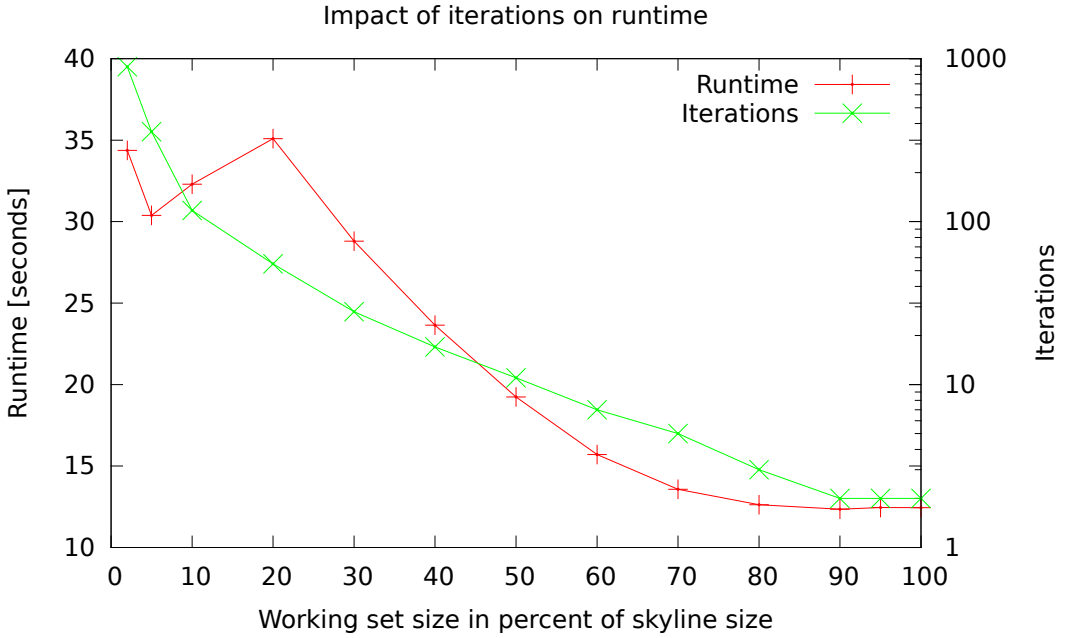


Figure 22: Influence of iterations on performance of Shifter List-based skyline implementation.

alternating between too much load and too little load.

7 Other algorithms

In this section we analyse which problems could be solved using a (possibly generalized) Shifter List implementation.

7.1 Frequent item

In a data stream, determining the n most frequently occurring elements is non-trivial. An accurate solution would require memory linear in the number of distinct items which occur in the stream as it needs to store the number occurrences for every item it saw so far.
As this is not desirable for a data stream with a potentially enormous number of distinct elements, approximation algorithms have been devised which provide certain error-guarantees for the most frequent items. One of these algorithms is called *Space-Saving*, developed by Metwally et al. in [15].

7.1.1 Space-Saving: Overview

Space-Saving provides a fixed amount of buckets to count the occurrences of frequent items. It accurately counts the occurrences for items until it encounters a new item and has no bucket to store it in.
Space-Saving now replaces the least-occurring item in its buckets e_{min} with the count $\zeta_{e_{min}}$ with the new item and assigns the new item the count $\zeta_{e_{min}+1}$. This results in a certain inaccuracy which is the trade-off for the space which is saved compared to the traditional approach.

One obvious property of Space-Saving is that its result depends on the exact permutation of the input due to its replacement strategy. Another is that the occurrences are only over-, never under-estimated.

7.1.2 General implementation

This implementation implements Space-Saving on top of Shifter Lists. In the local working set of each node we store the items and their number of occurrences in a descending sort order.
The items are streamed downwards and the smallest occurrence count of a node is passed along with the points to maintain the sort order between the nodes.
The transfer protocol will be changed to respect and maintain that order.

7.1.3 Sort order invariant

At each node, we maintain a set of items and occurrence counts. The algorithm will try to preserve the following order: $cmin_i \geq cmax_{i+1}$ This simply means that the nodes should try to respect a global descending order. As soon as node detects that this order is violated, it will initiate a transfer to re-establish it.

7.1.4 Transfer protocol

To transfer the occurrences between the nodes, we need an asynchronous transfer protocol which preserves the sorting invariant. For this, each node needs some awareness of its neighbours' state. A node forwarding an item sends along the minimum occurrence count of its working

set. A transfer is initiated when the node further downstream sees the invariant violated, i.e.: $cmin_i < cmax_{i+1}$

Transfer protocol

1. A node receives a point which matches an item in its working set. It updates the occurrence counter for the item and notices that the invariant is now violated: The point has a higher occurrence count than the smallest occurrence count in the adjacent upstream node.
2. The node sends a TRANSFER message to the upstream node with the point needed to re-establish the invariant and its occurrence counter. It stores a snapshot of the occurrence counter at time of the transfer message.
3. The upstream node receives the message and:
   (a) Accepts the point into the local working set, if the invariant is still violated. Should the local working set be too full, the item with the smallest occurrence is sent with a ACK message downwards. Otherwise an empty ACK is sent.
   (b) It rejects the transfer with a NACK message, if the invariant had been restored in the meantime by another processed item.
4. If the downstream node receives the ACK, it calculates the difference between the snapshot of the occurrence counter and the current state of it. It then sends this delta with a DELTA message to the upstream node. The element and its snapshot can now be removed from the working set of the downstream node.
5. The upstream node receives the DELTA message and adds the delta to the local occurrence counter.

The protocol requires a few additional conditions:

- A node can only take part in one transfer at a time.
- The last node must not accept input while waiting for an ACK. It must send up any input that it receives for reinsertion.[11]

The second conditions stems from the fact that there still would be insufficient space and that only one transfer can be in progress such as to maintain the sort order invariant. Note that this will cause a re-ordering and can change the output of a Space-Saving implementation.

7.1.5 Evaluation

This implementation of Space-Saving requires a significant amount of coordination in the transport protocol as well as additional state (the snapshot of the occurrence counters). Due to this overhead it is unlikely that the implementation will offer a superior performance compared to current implementations.

[11] It has to send it to the node upstream for reinsertion to prevent a deadlock which would occur if the message buffer for its inbound communications channel were to be full.

Roy et al. [17] presented an implementation where a single node was used to pre-filter the input to Space-Saving, resulting in a two-step pipeline. The implementation already significantly improved the throughput and filters out a significant percentage for even small distribution skews (Zipf-parameter).
To achieve an improvement over this implementation with a Shifter List implementation, the nodes would have to receive enough load, which is unlikely given that one node already pre-filters a significant part of the workload and that the frequent item problem is not as computationally-intensive (as compared to the skyline problem).

7.2 N-closest pairs of points

The n-closest pairs of points problem is the general form of the closest pair of points problem (n=1). It consists in determining the n pairs of points in a set with the minimal distance. For the special case n=1, there exist divide-and-conquer solutions to determine the closest pair of points in $O(n * log(n))$ [12].
The n-closest pairs of points problem should not be confused with the n-nearest neighbours problem in which for a fixed point the nearest n points are to be determined.

7.2.1 Definition

Assuming a set of k points $M = \{p_1, p_2, \ldots, p_k\}$, where all p_i are from the d-dimensional space T and a distance function $f : T \times T \to \mathbb{R}$. Then the n-closest pairs of points are a set S of n pairs $\langle p_i, p_l \rangle$ of points chosen from M, such that:

$$\forall \langle p_i, p_j \rangle \in S : p_i \neq p_j \tag{1}$$

$$\forall \langle p_i, p_j \rangle \in S : \quad \nexists \langle p_q, p_r \rangle \in S - \langle p_i, p_j \rangle : \quad (p_i = p_q \wedge p_j = p_r) \vee (p_i = p_r \wedge p_j = p_q) \tag{2}$$

$$\nexists \langle p_i, p_j \rangle, \quad p_i \neq p_j, p_i \in M, p_j \in M, \langle p_i, p_j \rangle \notin S : \quad f(p_i, p_j) < \left(\underset{\langle p_q, p_r \rangle \in S}{max} f(p_q, p_r) \right) \tag{3}$$

A point may only occur once in the same pair as specified by Equation 1 and there may not be duplicate pairs by Equation 2. Equation 3 specifies that no pair of points in M (that is not in the solution S) can have a smaller distance than the largest distance of a pair in the solution S.

7.2.2 General implementation

The implementation follows the implementation of the skyline algorithm, with a few adjustments. The first node parses the input and forwards the points downwards the input stream. The last node inserts them into its working set and transfers them to the upstream nodes.

Each node keeps the N pairs of points with the smallest distances encountered so far. Furthermore, each node has a working set to store points which it receives from its neighbour node downstream and transmits to its neighbour node upstream. It does not keep the points it receives from upstream but instantly compares these points to the points in its window and tries to store the calculated distance. An overview of this can be seen in Figure 23 and the adjusted processing algorithm in algorithm 11.

When all input has been processed, we merge the working sets of the nodes with a simple sorting algorithm in ascending pair distance and obtain the n closest pairs.

[12] see theorem 6: M. Shamos and D. Hoey, "Closest-point problems," Foundations of Computer Science, 1975., 16th Annual Symposium, pp.151-162

Algorithm 11 Processing of a datapoint (N-closest pairs of points)

```
1: procedure PROCESS(datapoint pt, node n)
2:     if pt == PT_NOP then
3:         return DROP                                  ▷ Can discard this immediately
4:     end if
5:     for datapoint c in n.candidates do
6:         if !c.valid then
7:             continue
8:         end if
9:         if pt.timestamp > c.timestamp then
10:            if c.valid and !(c.beingSent or c.beingReceived) then
11:                dropFromCandidates(c)
12:                continue
13:            end if
14:        end if
15:        d = distance(c,pt)
16:        insertDistance(d,n)          ▷ Adds the distance into the sorted set and trims size
17:    end for
18:    return FORWARD
19: end procedure
```

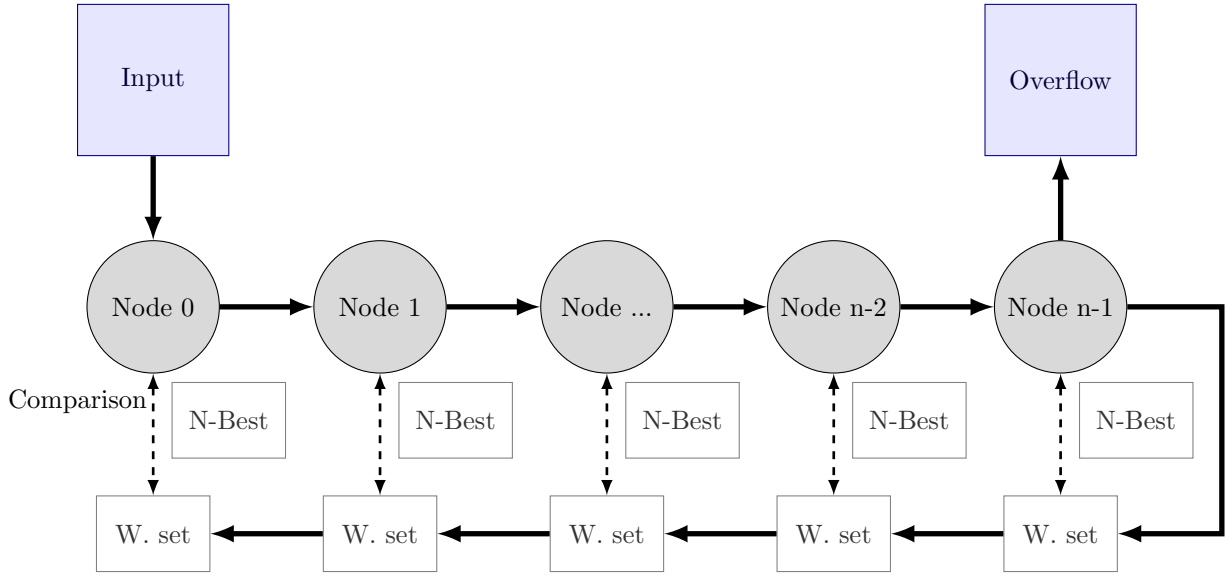


Figure 23: Flow of points (solid) and comparisons (dashed)

7.2.3 Transfer, overflow, timestamps

Because we only need to transfer points between nodes and no invalidation occurs, we can simplify the transfer protocol to a TRANSFER and an ACK message. The (in-)validation messages are not required. This implies that points that have been received in a transfer are valid immediately and can be immediately transferred again by the receiver.

We again use the same pattern of an overflow file and timestamps to ensure that points in the working sets have been compared to all other points. Unlike in the skyline algorithm, we then drop these entirely from the window instead of putting them into the output set.
The conditions and mechanisms for timestamping are identical to the skyline implementation.

7.2.4 Limitations

The implementation uses significantly more memory than a single-threaded naive version: The sum of the working set sizes amounts to $n \cdot \#nodes$ as each node needs to store the n-best results.

7.2.5 Optimizations

The eager accept optimization from algorithm 10 can be applied. Further, to remove the memory overhead mention in the previous subsection, one could be to enforce a global ordering over the best distances kept at each node, but this is beyond the scope of this thesis.

7.3 Sorting

Sorting a set of elements according to a specified order is one of the most common problems in computer science, thus we omit the formal definition here. This subsection details how it can be implemented using a Shifter List-based implementation.

7.3.1 General implementation

The points are read and forwarded by the first node. The invariant of this implementation is that every node keeps its working set sorted according to the defined order. We assume here that the sort order is ascending.

Each node compares the received points to their working set. It will insert the point if it is smaller than the largest one in its working set or the working set is not full. Otherwise it is forwarded to the next node.

Algorithm algorithm 12 details the processing of a point, assuming that the chosen sort order is ascending. Note that the implementation does not require the transfer protocol because points are only forwarded downstream.

If the last node cannot accept all elements into the working set, they will be stored in the overflow. If the input is empty, a cycle is complete and all points in the working set can be output and then removed from the working set. If the overflow is not empty, all working sets are flushed and the overflow becomes the new input.

If the sum of the working sets is K, then in every such cycle, the K-smallest points of the remaining input will be stored in the working sets.

Algorithm 12 Processing of a datapoint (Sorting)

```
1: procedure PROCESS(datapoint pt, node n)
2:     if !n.candidates.full then
3:         putIntoCandidates(pt)    ▷ This method will insert the point and keep the set sorted
4:         return DROP
5:     else if n.candidates.last > pt then
6:         return FORWARD
7:     else
8:         last = popLastCandidate(n)                          ▷ Last one is the largest one
9:         putIntoCandidates(n, pt)
10:        forward(last)
11:        return DROP
12:    end if
13: end procedure
```

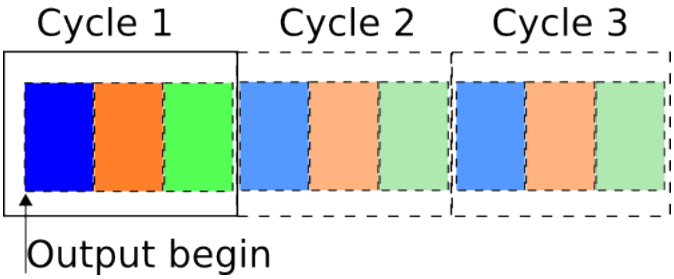


Figure 24: Output pattern for sorting

7.3.2 Properties

This implementation establishes a sort order among the nodes. Assuming an ascending sort order, the last element of any node's working set will be smaller or equal than the first element of the next downstream node's working set:

$$n_i.last \leq n_{i+1}.first$$

Also note that the size of a node's working set can only increase or remain constant (between comparisons) because elements are never dropped and only forwarded if there is insufficient space to insert a smaller element.

7.3.3 Output

Because of the locally sorted working sets and the sorting property mentioned above, the result is created by concatenating the working sets of the nodes in node order. This can be implemented lock-free by assigning every node the area into which it shall copy its local working set.
If the sum of the working set sizes of each node is smaller than the input size, multiple cycles are needed. After every cycle, the output area for each node advances exactly by the sum of the local working sets (see Figure 24).

7.3.4 Limitations

Each node has to keep its working set ordered in this implementation, thus insertions and deletions need to preserve this invariant.
In an array implementation, locating the insertion point can be done in $O(log(n))$. However, because the elements have to be moved to allow the insertion, insertion which does not happen at the end of the array has linear complexity!
In a linked list implementation of the sorted working set, insertion and deletion are possible in constant time. Locating the insertion point however, requires traversal of the data structure as binary search is not possible (no random access) and thus exhibits linear complexity as well.

7.3.5 Evaluation

Because of the expensive preservation of the local sorting order in the working sets, this implementation of sorting will likely not perform well against other established sorting methods. Tentative experiments with quicksort confirmed this impression.

7.4 Problem properties

For a problem to be solvable with a Shifter List based implementation, it needs to meet the following conditions:

1. It needs to be splittable into independent sub-elements.
2. The sub-elements must contain all necessary state. There cannot be a global state or shared state between sub-elements.
3. It has a computational complexity of at most $O(n^2)$, where n is the number of sub-elements.
4. Sub-elements may not influence the state of other sub-elements, unless they are processed together.
5. The result of processing sub-elements must only depend on these sub-elements and their state.

Condition 1 is required to be able to process the problem in a pipelined approach. Because a Shifter List node cannot access the state of other nodes, all state needs to be in the sub-elements (Condition 2). As we can at most compare all sub-elements with each other by adding them to the working sets, this limits us to problems with quadratic complexity or less (Condition 3). Condition 4 and 5 are dictated by the independence of nodes. Note that points are "processed together" if they are either present in the working set or taken for processing from the down- or upstream.

7.4.1 Problems

We show how two of the presented problems fulfil the conditions set above (Skyline and Frequent item) and can be implemented with a Shifter List-based implementation and one problem that does not fulfil the conditions (Shortest Path).

Skyline

1. Every point in the input set is a sub-element.
2. The state of a sub-element is the timestamp and the association to a node's working set. No global state is required.
3. Comparing all sub-elements with each other is sufficient to determine the result (Naive algorithm).
4. Sub-elements only interact with each other on comparison.
5. The processing of sub-elements only depends on the involved sub-elements (dominance test) and their state (timestamp).

Frequent Item

1. Every occurrence of a term is a sub-element
2. The terms contain their term and the number of occurrences (initially 1). No global state is required
3. Comparing all sub-elements with each other would allow grouping identical ones into buckets and counting them. Sorting after the main shifter algorithm would only be $O(b \cdot log(b))$ (b being the number of unique terms), well inside the quadratic complexity bound because $b \leq n$.
4. Sub-elements only influence each other when compared. (One sub-element has its occurrence counter incremented and the sub-element is dropped, if they have an identical term.)
5. The processing of sub-elements only depends on the involved sub-elements and their state (term, occurrence count).

Counterexample: Shortest Path Determining the shortest path between two vertices in a graph (typically solved using Dijkstra's algorithm) is not possible with this template. The problem has the following properties:

1. Every vertex is a sub-element.
2. Every sub-element contains the distances to other sub-elements. There must be a global table with the minimum distance for each node. ϟ
3. Comparing all elements with each other would allow to build the paths and discover the shortest path in quadratic complexity of the sub-elements.
4. Sub-elements only influence each other when compared.
5. The processing of two sub-elements only depends on the state of these two and the global minimum-distance table for the sub-elements. ϟ

As we can see, the minimum-distance that vertices would need to store in a global manner violates condition 2. Also, for processing, this global table would be required, which violates condition 5. Thus this problem cannot be implemented in a Shifter List.

8 Shifter List template

In the previous sections we have seen different ways to implement algorithms based on a Shifter List structure. In this section, we detail a common template for implementations based on the Shifter List structure and how some of the algorithms detailed before can be implemented with this template.

8.1 Design

The template is implemented in C and thus we chose to implement the delegate calls with function pointers. It contains several compile-time options that can be used to enable debugging, statistics or correctness checks.

The main design approach behind the template was the delegation pattern: The general Shifter List provides a implementation for passing data between nodes and issues a series of delegation calls. The actual algorithm can then decide what actions to take for the items provided by the delegation call.

Certain methods are exposed to easily allow delegate functions to manipulate the state of a node or communicate with adjacent nodes. The exposed methods above operate with the Shifter List concepts of nodes. These are actual structures which hide additional implementation details. This is an application of the concept of information hiding.

8.1.1 Configuration

The execution is started by calling an execution function with a configuration structure. The configuration must contain pointers to all necessary delegates (or to stubs[13]) as well as specifying the input and any options such as the number of threads to be used.

Furthermore, the implementation has to configure the behaviour of the template at the at the end of an iteration: If it should terminate, or swap the overflow with the input and start a new iteration. In the latter case, the template will iterate as long as there still is input to process.
The type of the data elements (floating point, integer or other) can be changed by changing a `typedef` in a single header file.

8.1.2 Data flow & Transfer

By default, the template will send tuples downstream without a special protocol and use a simple two-step transfer protocol (send, ACK/NACK) to transfer data elements upstream. The default behaviour can be extended or overridden with delegate functions (see subsubsection 8.1.4).

8.1.3 Data points and state

Every data point contains the payload, an identifier for the point, a timestamp and three flags. Of these flags, one is a validity flag and two are used to indicate if the point is in a transfer (beingSent, beingReceived).

The node additionally uses a state flag to cache if it is the sending node in a transfer. This saves the iteration over the working set's beginSent-flags.

[13] An empty or default implementation of a method.

The working set of each node is configured with a maximum size. This value can be overridden at runtime by providing an initialization delegate. The defaults are determined by a set of compile time constants: The initial working set size, the scale factor and the maximum size.

8.1.4 Delegation calls

The following calls occur in the context of a node and therefore always pass along the node as an argument.

- **onNodeInit** (Optional) When the setup is being performed, this method may perform custom initialization of the node state. E.g. Set the maximum size of a node's local working set.
- **onProcessData** (Required) When a message with data points is received, the template will issue a delegate call to process each data point. The function delegate can perform any action on the data point and must then indicate what should happen with it: Either it can be dropped or it can be forwarded to the next node.
- **onCycleEnd** (Optional) When the input has completed one iteration and has been processed or stored in the overflow, this function delegate is called.
- **onLoopEnd** (Optional) As the template loops indefinitely and waits for input, this delegate is called once per loop iteration.
- **onTerminate** (Optional) This delegate is called when all input has been processed and no input remains. After this delegate call, the node will receive no more delegate calls.
- **onTransferBegin** (Optional) This delegate function is called to fill a message with all elements that should be sent. The default implementation will try to fill the message with as many elements from the local data set as possible.
- **onTransferReceived** (Optional) Upon receipt of the transfer, this delegate function is called. It can decide what to do with the received elements and return a message to the template as answer (ACK, NACK or custom). It receives the upper bound of the elements that should be provided as argument. The default implementation will accept the elements into the local working set if there is enough space and return an ACK, resp. a NACK if there is not.
- **onTransferComplete** (Optional) This delegate function is called when the sending node receives an ACK message. The delegate function can then send additional custom protocol messages. The return value indicates to the template if the transfer should be regarded as complete (and involved elements deleted from the local working set) or if the custom implementation will take care of further actions. NACK messages will be handled automatically and the transfer is aborted without a call to this delegate.
- **onProcessSpecialMessage** (Required) This delegate function is used when the Shifter List template does not recognize the type of a message that has been received by this node. This can be used for custom messages, such as transfer protocol extensions.

If the implementation provides the delegates for the transfer methods, it will need to make sure that the protocol is sound and that all state flags of the data elements are properly maintained.

8.1.5 Output

Unless an output memory pointer is provided, the template will allocate enough memory to store as many output elements as input elements.
The template provides a thread-safe output (with arbitrary order) into this memory region with an output function that the implementation may use. If special ordering is required, the implementation must provide this itself.

8.1.6 Exposed methods

The template exposes the following methods to allow the algorithm to modify the state of a node or communicate with neighbouring nodes.

- **sendMessageUp/-Down** sends a message to the appropriate node
- **add-/removeFromWorkingSet** tries to add/remove a point to the data set. For insertion, it accepts an optional parameter indicating the insertion position in the set.
- **sendWorkingSetUp** initiates a transfer. It uses the `onTransferBegin` delegate if available to determine the elements that should be transferred. It accepts an upper bound for the number of elements.
- **canOutput** checks the flags to see if the data point is not in any transfer.
- **output** will output the point to the memory area.

8.1.7 Limitations

An generic implementation using such delegates (function pointers) cannot perform as well as an implementation targeted at a single algorithm which allows for more optimizations.
Further, the template does not provide built-in support for a transport protocol that forwards tuples downstream. However, such a protocol can easily be added by using the appropriate delegates and custom messages.
Some extensions (e.g. extending the global state data structure of the algorithm) require extensions of the data structures defined by the template, while others can make use of prepared custom state fields.

8.2 Skyline

This subsection will detail the configuration used to run Skyline computations using the template presented.

8.2.1 Configuration

The template was configured to restart when the initial input has been read (and remainder of the input is in the overflow). We do not need to configure a `onTransferBegin` or `onTransferReceived` delegate as the default will suffice.

8.2.2 Modification

The skyline requires two additional states per node: A flag indicating if the working set of the node has already once been full (related to subsubsection 5.2.4) and a field to store how much space the upstream node has for a transfer. For this we just added the two fields to the struct definition.

8.2.3 Processing

The processing delegate basically implements algorithm 6, with a small extension that will check if a HAS_SPACE message was received and initiate a transfer if that was the case. When initiating a transfer by calling `sendWorkingSetUp`, the maximum number of elements to be set is read from the stored value initially received in the HAS_SPACE message.

8.2.4 Custom Messages and Transfer

The messages for indicating if the data points that were transferred are valid or not, we need the two custom messages VALID and INVALID, and thus a `onProcessSpecialMessage` delegate. The delegate will appropriately delete invalidated transferred points (INVALID case) and mark the remaining points as valid and remove the transfer flag. Further, this delegate must also handle the HAS_SPACE message and store the contained data in the node state.
To actually perform the validation and send the VALID/INVALID messages, we need to install a `onTransferComplete` delegate.

8.2.5 Other delegates

To transfer information about the available space for transfers, we use a `onLoopEnd` delegate and send a HAS_SPACE message when needed. Further, we want to output all remaining elements in the working set when the template terminates which our `onTerminate` delegate will perform.

8.2.6 Evaluation

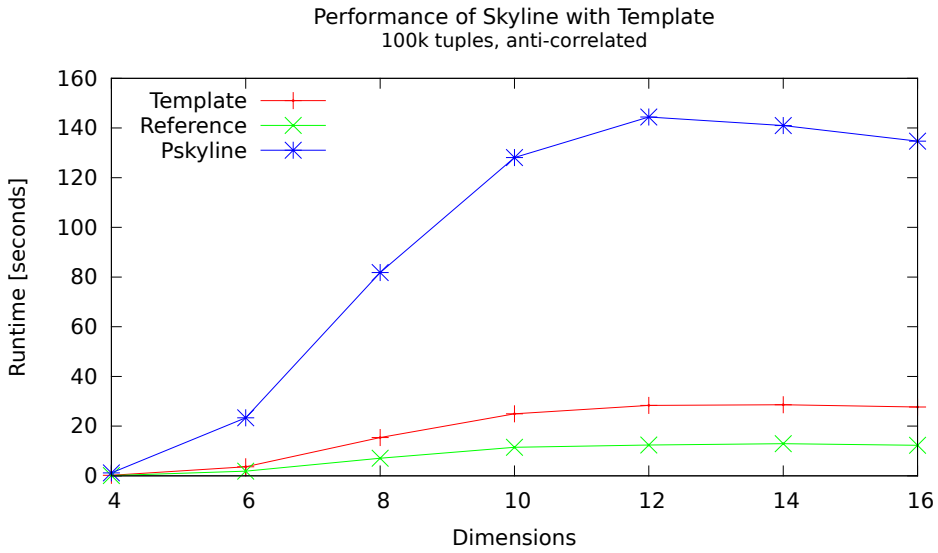
We implemented this algorithm with our template and checked for correctness of the results. Further, we re-ran two experiments for the template-based implementation to examine the behaviour and performance.

In Figure 25, we plotted the performance of the template when the number of dimensions (and thus the size of the skyline) is varied. We can see that for both the uniform and anti-correlated dataset, the behaviour of the template is slower, but follows the same behaviour as the reference implementation developed in section 5. However, the template implementation still outperforms *pskyline*.
In Figure 26 we plotted the result of re-running the scalability evaluation of the template. We can that the behaviour of the template matches that of the reference implementation, though it does not exhibit the small drop at 28/32 nodes.

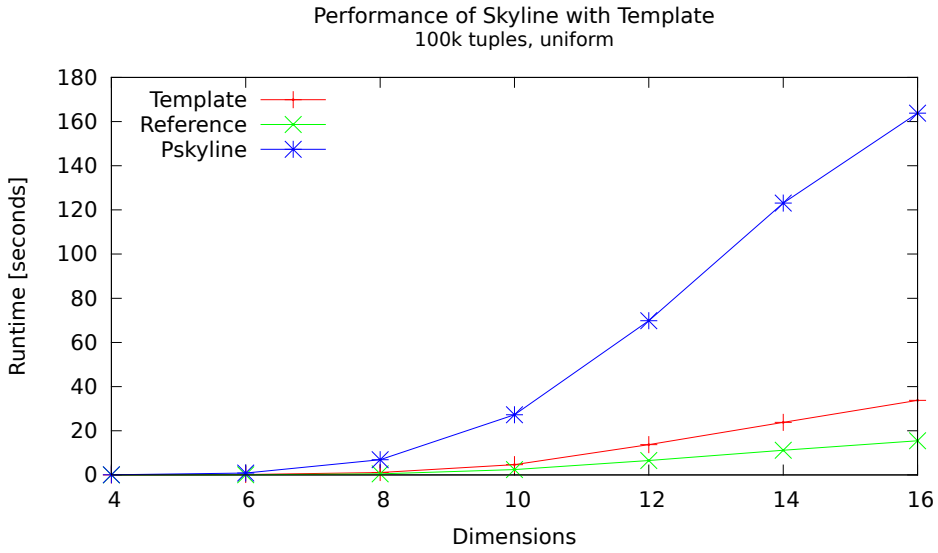
We can thus say that the template exhibits most of the relevant behaviour of the reference implementation, though it has a lower performance.

8.3 Sorting

To implement sorting with the shifter list template, we only need three delegates: The mandatory processing delegate as well as the `onCycleEnd` and `onTerminate`. We configured the template to restart on the end of an iteration. We do not need the transfer delegates as no transfer is required in this implementation. We implemented this algorithm with the template and affirmed its correctness.



(a) Performance of skyline template on anti-correlated dataset



(b) Performance of skyline template on a uniform dataset

Figure 25: Performance comparison of Skyline implementation with / without template

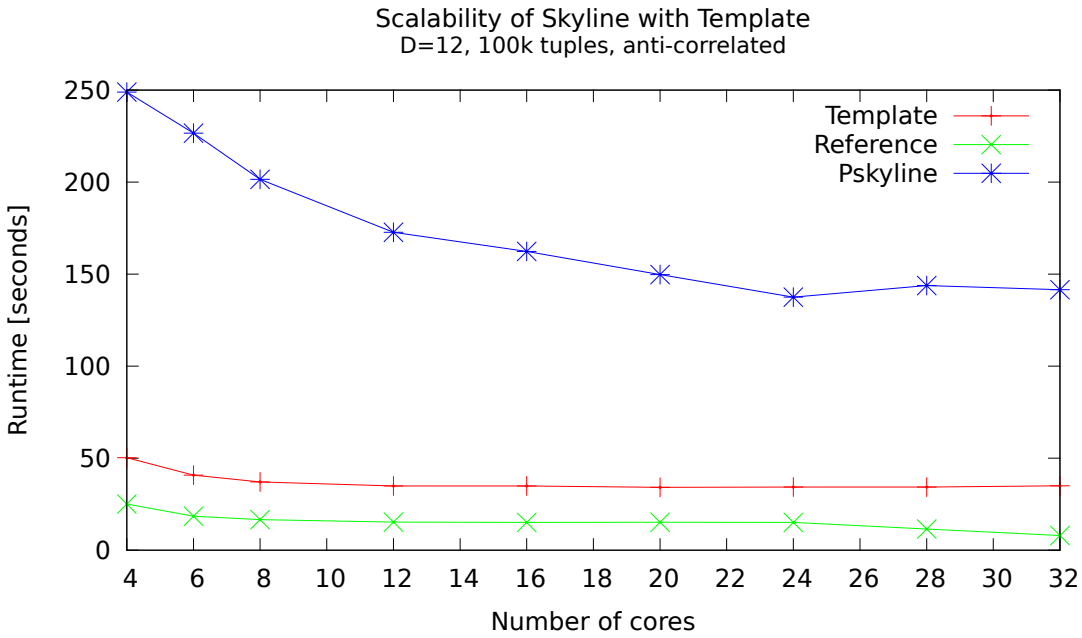


Figure 26: Scalability comparison of Skyline implementation with / without template

8.3.1 Modification

For output, each node needs two pieces of information: A pointer to his initial output area and an offset (the sum of all working set sizes) so that he can calculate the next output position after a cycle ended. This information has to be added in the configuration.

8.3.2 Delegates

The `onProcessData` delegate inserts the data into the local data set and preserve the sort order and possibly forwards elements to the downstream node. The `onCycleEnd` delegate outputs the data elements in the local working set and calculates the new output position for the node in the next iteration. The and `onTerminate` delegate is identical, but does not require the calculation of the next output position. For simplicity however, only one implementation needs to be created and both delegates can be configure to run the identical implementation.
As these output areas are exclusive to the nodes, no synchronization for output is needed.

8.4 Discussion

We have demonstrated in this section how two problems can be solved with a Shifter List-based implementation. We have shown that the overall behaviour of the template implementation of solving skyline queries corresponds with the non-template version developed in this thesis.
As expected, the performance of the template-based implementation is lower than its reference implementation. This is due to the generic nature of the template which incurs some performance penalties for using delegate calls and does not benefit from the whole optimization range

as compared to the reference implementation.

Nonetheless, the template is a very helpful: The time to develop an algorithm with the template and see first performance results (which in the case of skyline still beat the competitor) is significantly lower than implementing the whole algorithm from scratch. A first implementation can verify the approach and general behaviour with the template, while for an optimized version the delegate calls can be inlined and further optimized.

9 Conclusion

We have presented a new approach using pipeline-parallelism to solve skyline queries on multi-core architectures. We have shown that it outperforms a state-of-the-art parallel implementation which is based on input partitioning (*pskyline*, Park et al. [12]) by an order of magnitude, except on datasets with a very small skyline. On datasets where the size of the skyline is very small compared to the input size, a simple single-threaded approach already achieves a very high performance as it can efficiently discard most input tuples. Parallelized approaches are not necessary for such datasets.

In our evaluation, the experimental results have shown that while our implementation is affected by skew (with respect to the position of skyline elements within the dataset) too, the effect is smaller than for the compared algorithm. Additionally we observed that the configuration factors significantly influence the performance of our approach.
Furthermore, we presented a template based on the Shifter List concept to implement algorithms and showed how it can be used to implement skyline queries and sorting. The template allows to quickly implement an algorithm with a pipelined approach, while avoiding a large part of the tedious boilerplate-code. While the performance is not as high as with an implementation that is focused on one algorithm, it allows for a quick evaluation of the viability and correctness of an approach. It offers a performance baseline from which optimizations can be implemented. Though slightly slower, the template skyline implementation still outperforms *pskyline* by a large margin.

A Appendix

A.1 FPGA

An FPGA (field programmable gate array) is a hardware circuit which can reconfigure its pathways at initialization or runtime. It can be used to implement almost arbitrary operations by properly routing signals and configuring its components.
Typically an FPGA consists of interconnection wires (with contain special re-configurable junctions/intersections), lookup tables (LUT) to implement arithmetical or logical operations and very small memory blocks (BRAM).

When implementing a circuit on an FPGA, the designer of the circuit is limited by the number of these resources. Signal propagation delays on the other hand restrict the maximum clock frequency that the circuit run. Also, the layout and routing of the wires is very complex and becomes harder the larger the size of the circuit is. Thus circuit design is typically done in a hardware description language such as VHDL or Verilog. The actual routing and resource allocation is then done by a vendor-specific tool which takes the resource constraints of the FPGA

node	0	1	2	3	4	5	6	7
0	10	16	16	22	16	22	16	22
1	16	10	22	16	16	22	22	16
2	16	22	10	16	16	16	16	16
3	22	16	16	10	16	16	22	22
4	16	16	16	16	10	16	16	22
5	22	22	16	16	16	10	22	16
6	16	22	16	22	16	22	10	16
7	22	16	16	22	22	16	16	10

Table 6: Node distances obtained by executing *numactl -H* on a 4-socket machine with AMD Opterons 6276 (Note that the AMD Opteron 6276 consists of two processors 'fused' on a single die - each with its own memory controller. Thus on a machine with four sockets, we can see eight NUMA nodes.)

into account.

The fact that the number of clock cycles that operations take are known in advance can be exploited in the design of a parallelized implementation to avoid synchronization or waiting that would be normally required on normal hardware with a pre-emptive scheduler .

A.2 Non uniform memory access (NUMA)

In every computer, reading from or writing to memory by the processor takes a certain amount of clock cycles. The number of these cycles usually depend on the architecture and hardware, but also on whether the data is or can be cached.
For a long time, the whole area of main memory could be accessed by the processor within the same number of cycles (assuming any involved caches miss).
In NUMA-systems this is no longer the case: Not all memory regions are accessible by a processor at the same cost: "The memory access time depends on the memory location relative to the processor"[14]. For certain implementations, a processor may not even have access to certain memory regions and has to request it from the owning processor. This leads to an increased access time. The tool 'numactl'[15] can print the distances between NUMA nodes. An example is given in Table 6

Placement and access patterns for data thus can influence performance of a memory-intensive, multi-threaded program (see e.g. [16]).

[14] Non-Uniform Memory Access, Wikipedia, retrieved on 2013-08-13
[15] Requires a NUMA-aware Linux kernel. (2.5 and up: see [21], chapter 2)

References

[1] H.T. Kung, P. Luccio, and F.P. Preparata, "On Finding the Maxima of a Set of Vectors", ACM Vol. 22, No. 4, 1975, pp.459-476

[2] S. Börzsönyi, D. Kossmann and K. Stocker, "The Skyline Operator" in ICDE 2001, pp. 421-430

[3] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: an online algorithm for skyline queries", in VLDB 2002, pp.275-286.

[4] J. Chomicki, P. Godfrey, and J. Gryz and D. Liang, "Skyline with Presorting", 2002

[5] D. Papadias, Y. Tao, G. Fu, B. Seeger, "Progressive skyline computation in database systems", ACM transactions on database systems, Vol. 30, No. 1, 2005, pp.41-82

[6] K. Lee, B. Zheng, H. Li and W. Lee, "Approaching the Skyline in Z Order" in VLDB 2007, pp.279-290

[7] P. Godfrey, R. Shipley and J. Gryz, "Maximal Vector Computation in Large Data Sets" in VLDB 2005, pp. 229-240

[8] S.-R. Cho, J. Lee, S.-W. Hwang, H. Han, and S.-W. Lee. "VSkyline: vectorization for efficient skyline computation", SIGMOD Rec. Vol. 39 No. 2, December 2010, pp.19-26

[9] B. Cui, H. Lu, Q. Xu, L. Chen, Y. Dai, and Y. Zhou, "Parallel Distributed Processing of Constrained Skyline Queries by Filtering", ICDE 2008, pp.546-555

[10] A. Vlachou, C. Doulkeridis, and Y. Kotidis, "Angle-based space partitioning for efficient parallel skyline computation", Proceedings of the 2008 ACM SIGMOD international conference on management of data, pp.227-238.

[11] H. Köhler, J. Yang, and X. Zhou, "Efficient parallel skyline processing using hyperplane projections", Proceedings of the 2011 ACM SIGMOD international conference on management of data, pp.85-96

[12] S. Park, T. Kim, J. Park, J. Kim and H. Im, "Parallel Skyline Computation on Multicore Architectures" in ICDE 2009, pp. 760-771

[13] J. Teubner, and René Müller, "How soccer players would do stream joins", Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pp.625-636

[14] L. Woods, G. Alonso and J. Teubner, "Parallel Computation of Skyline Queries" in FCCM 2013, pp.1-8

[15] A. Metwally, D. Agrawal and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams", in ICDT 2005, pp.398-412. Springer Berlin Heidelberg, 2005.

[16] D. Tam, R. Azimi and M. Stumm, "Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors" in EuroSys 2007, pp.47-58

[17] P. Roy, J. Teubner and G. Alonso, "Efficient frequent item counting in multi-core hardware", 18th ACM SIGKDD international conference on knowledge discovery and data mining, 2012

[18] Advanced Micro Devices, Inc., "Performance Guidelines for AMD Athlon 64 and AMD Opteron ccNUMA Multiprocessor Systems", http://support.amd.com/us/Processor_TechDocs/40555.pdf, retrieved on 2013-06-04

[19] Advanced Micro Devices, Inc., "AMD Opteron 6200 Series Processors Linux Tuning Guide", http://developer.amd.com/wordpress/media/2012/10/51803A_OpteronLinuxTuningGuide_SCREEN.pdf , retrieved on 2013-07-15

[20] Y. Li, I. Pandis, R. Mueller, V. Raman and G. Lohman, "NUMA-aware algorithms: the case of data shuffling" in CIDR 2013

[21] M. Dobson, P. Gaughen and M. Hohnbaum, "Linux Support for NUMA Hardware", Proceedings of the Linux Symposium 2003, https://www.kernel.org/doc/ols/2003/ols2003-pages-169-184.pdf, retrieved on 2013-06-07